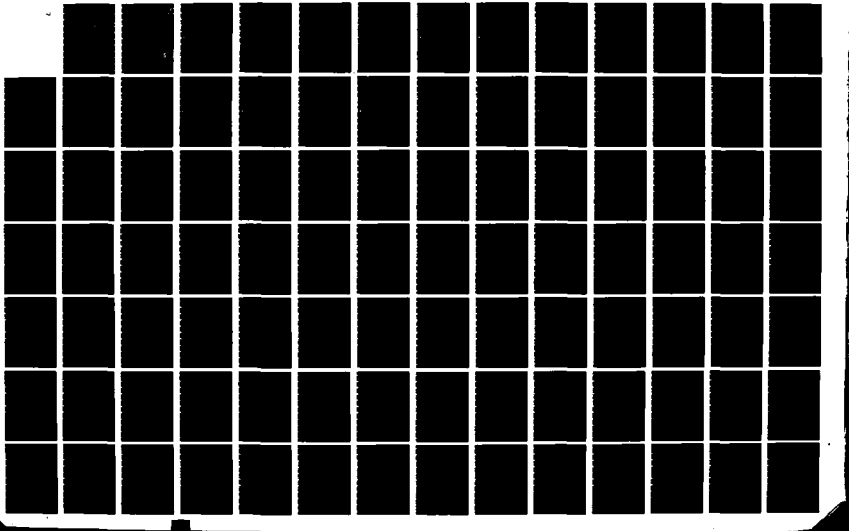
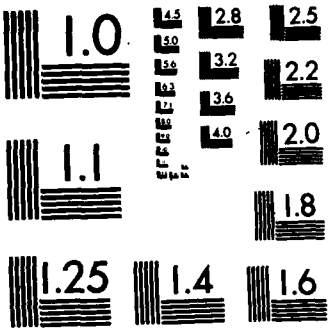
AD-A138 083 DETAILED DESIGN AND IMPLEMENTATION OF A MULTIPROGRAMMING OPERATING SYSTEM.. (U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. 1/3

UNCLASSIFIED P E CRUSER DEC 83 AFIT/GCS/EE/83D-5 F/G 9/2 NL





DETAILED DESIGN AND IMPLEMENTATION OF A MULTIPROGRAMMING OPERATING SYSTEM FOR SIXTEEN-BIT MICROPROCESSORS

THESIS

AFIT/GCS/EE/83D-5 Paul E. Cruser
2Lt USAF

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY (ATC)
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/EE/83D-5

DETAILED DESIGN AND IMPLEMENTATION OF A MULTIPROGRAMMING OPERATING SYSTEM FOR SIXTEEN-BIT MICROPROCESSORS

THESIS

AFIT/GCS/EE/83D-5 Paul E. Cruser
2Lt USAF

Approved for public release; distribution unlimited.

AFIT/GCS/EE/83D-5

DETAILED DESIGN AND IMPLEMENTATION OF A MULTIPROGRAMMING OPERATING SYSTEM FOR SIXTEEN-BIT MICROPROCESSORS

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

by
Paul E. Cruser, B.S.
2Lt USAF
Graduate Computer Systems
December 1983

Preface

This thesis presents the detailed design and implementation of a multiprogramming operating system for sixteen-bit microprocessors. The detailed design is based on the works of Robert J. Yusko, Mitchell S. Ross, and Douglas S. Huneycutt, Sr. I would like to thank these men for their efforts which made this effort possible.

I would like to thank my faculty advisor, Dr. Gary B. Lamont, for his advice that was given throughout this effort.

I would like to express my appreciation to Lt. Ronald K. Miller for his cooperation in this effort. Also, I thank Linda W. Miller for her cooking and patience.

Finally, I would like to thank my parents, Gerald D. and Molly A. Cruser, for helping me through college in more ways than one.

Table of Contents

List of Figures

Abstract

A multiprogramming operating system, designated AFIT Multiprogramming Operating System (AMOS), for the AFIT Digital Engineering Laboratory was designed at the detailed level and fully implemented, except for the assembly language routines. The requirements were developed in the works of Yusko, Ross, and Huneycutt.

This thesis effort was done in conjunction with the effort of Lt. Ronald K. Miller. This effort covers the detailed design and implementation of the overall system and, also, covers the detailed design and implementation of the operating system scheduler.

I. Scope of Project

Introduction

The purpose of this project is to continue the design and implementation of a multiprocessing operating system for sixteen-bit microcomputers. This operating system will be referred to as the AFIT Multiproramming Operating System (AMOS) (Ref. 1:1). The purpose of this chapter is to give a brief overview of operating systems, to outline requirements for AMOS that have been defined in the previous efforts by Ross (Ref. 4), Yusko (Ref. 5), and Huneycutt (Ref. 1), to outline the objectives of this project, and to present the approach to obtain the stated objectives.

One definition of an operating system (O/S) is "an organized collection of systems or programs that acts as an interface between machine hardware and users, providing users with a set of facilities to simplify the design, coding, debugging, and maintenance of programs while, at the same time, controlling the allocation of resources to assure efficient operation." (Ref. 2:1,2) In other words, the O/S is a large software management program that acts as an interface between the user and the computer system.

The computer system could include the hardware, application programs, and control.

Historical

In the first generation of vacuum-tubed hardware, the procedure for the operating system, which was nothing more than a program loader, was: a loader reads in an assembler; the assembler assembles into absolute code user source programs and library routines; the assembled code is written on tape or cards, and a loader is again used to read these into main storage; the absolute code of the program is then executed (Ref. 2:7). This meant that there was only one job executed at a time.

In the second generation of transistorized hardware, the operating system was developed into a sequential batch processing operating system. The operating system made use of new data channels, interrupts and used auxiliary storage efficiently (Ref. 2:9). This type of operating system still only allows one job to be executed at a time.

Along with the integrated circuitry of the third generation came the multiprogramming and time-sharing methods that could be used to make an operating system more efficient (Ref. 2:12). Multiprogramming is based on the concept of concurrency; that is, more than one program can be executing within the computer system at the same

time (Ref. 3:29). Since only one central processor is used in this type of environment, only a single program may be executing at a given instant in the central processing unit (CPU), but to the users it seems as if all the programs are executing at the same time. This is done with the use of input/output processing. Multiprogramming systems alternate the programs' usage of the CPU according to some policy. The operating system determines which program is ready for execution and then allocates the CPU for the program.

Time-sharing systems are an attempt to give each user a personal computer while efficiently utilizing the resources of a relatively expensive machine. All user interactions on a time-sharing system are done through on-line terminals. Two requirements for a time-sharing system are 1) the response time has to be maintained at the appropriate level of the human attention span and 2) the appearance of unrestricted access is presented to the user (Ref. 3:29).

The concepts of multiprogramming and time-sharing are complementary. Most minicomputer systems couple multiprogramming capabilities with an interactive time-sharing capability (Ref. 3:30). An example of such an operating system is UNIX (Ref. 20).

There are many types of operating systems on the market today for all sizes of computers. They range from

the simple batch to the complex network. These operating systems are more complex than the original ones and will become more complex in the future when greater needs are pressed on the operating system.

Review of Previous Requirements

This thesis effort is a follow-on to three previous thesis efforts that were under the direction of Professor Gary B. Lamont. Ross (Ref. 4) and Yusko (Ref. 5) were concerned with the upper level design of the operating system. Huneycutt (Ref. 1) was concerned with design and implementation of the file management of the operating system.

The previous requirements (given by Ross, Yusko, and Huneycutt) are followed in the final design and initial implementation of the operating system. Eight requirements for AMOS that are the goals for the initial implementation are:

1. Multiuser support for at least four concurrent users.
2. Friendly user interface.
3. Interuser communication.
4. Fair allocation of system resources.
5. Meaningful error diagnostics.

6. Recovery routine.
7. Minimal device/user utility support (Ref 1:11).
8. Provide a general purpose configurable O/S with full documentation to aid in teaching of O/S courses.

Although the Intel 8086 microprocessor was initially chosen (Ref. 4,5), the Z8000 was selected (Ref. 1) because it offered the desired support that was not provided by the 8080. The Z8000 is also capable of discerning between system and user tasks and can control the operations being performed for the users (Ref. 1:16). The choice of the Z8000 will be covered more thoroughly in the requirements chapter (Chapter 2).

The implementation of AMOS will be written mostly in the C language. Hardware dependent routines will be written in the Z8000 Assembly language. The reasons for using C are:

1. It is a structured language.
2. There is C source code for an existing operating system (UNIX) (Ref. 20) that is readily available to research.
3. The C language is less restrictive than other high-level languages that have an available compiler.

The modules that are to be written in assembly language will have to be rewritten in the new host computer's assembly language.

Objectives

The objective of this project is to design and implement a multiprogramming operating system for a sixteen-bit microprocessor. Top-down methodology is the main tool for design and implementation. The implementation of AMOS will be done in the C language, as stated earlier, and will avoid hardware configuration dependency (enhance portability). The only hardware dependent modules are those written in assembly language.

Approach

The project started with a literature search and review to obtain information on operating systems and their development. The requirements, for the most part are taken from References 1, 4, and 5. The structure of AMOS will be modular to facilitate testing, maintenance, and portability.

AMOS will be initially implemented on the Multibus Z8000 system (using the Z8002 microprocessor) from

Advanced Micro Devices (AMD). The Z8002 system contains a non-seqmented CPU card, a multi-port serial I/O card, 128K of main memory, a floppy disk controller, a clock/timer card, and a mainframe. This is the initial hardware that is present in the lab (Room 67, Building 640). More details on the Z8002 is contained in Appendix A.

Thesis Outline

The rest of the chapters will cover the following subjects:

1. Requirements for AMOS
2. Design of AMOS
3. Design and Implementation of AMOS Processor Manager
4. Implementation of AMOS
5. Conclusion and Recommendations

Requirements for AMOS, Design of AMOS, and Implementation of AMOS are duplicated in Lt. Ronald K. Miller's thesis document. All of the appendices are also duplicated.

II. Requirements

Introduction

The development cycle of a software project begins with the requirements analysis (Ref 9:198). The broad requirements for this project's operating system can be stated in one sentence. The operating system is to be a multiprogramming operating system for a sixteen-bit microprocessor computer system that is easily changed and is machine dependent only at the lowest levels. The next phases (Ref. 9: 199) of the software development cycle are:

1. Specifications/Requirements
2. Design
    a. Structural design
    b. Detailed design (algorithms)
3. Implementation/Coding
4. Testing
5. Operation/Maintenance

The steps are shown graphically in Figure II-1 (Ref. 11: 13). Although the testing is listed as the fourth phase, it should be done throughout the development cycle.

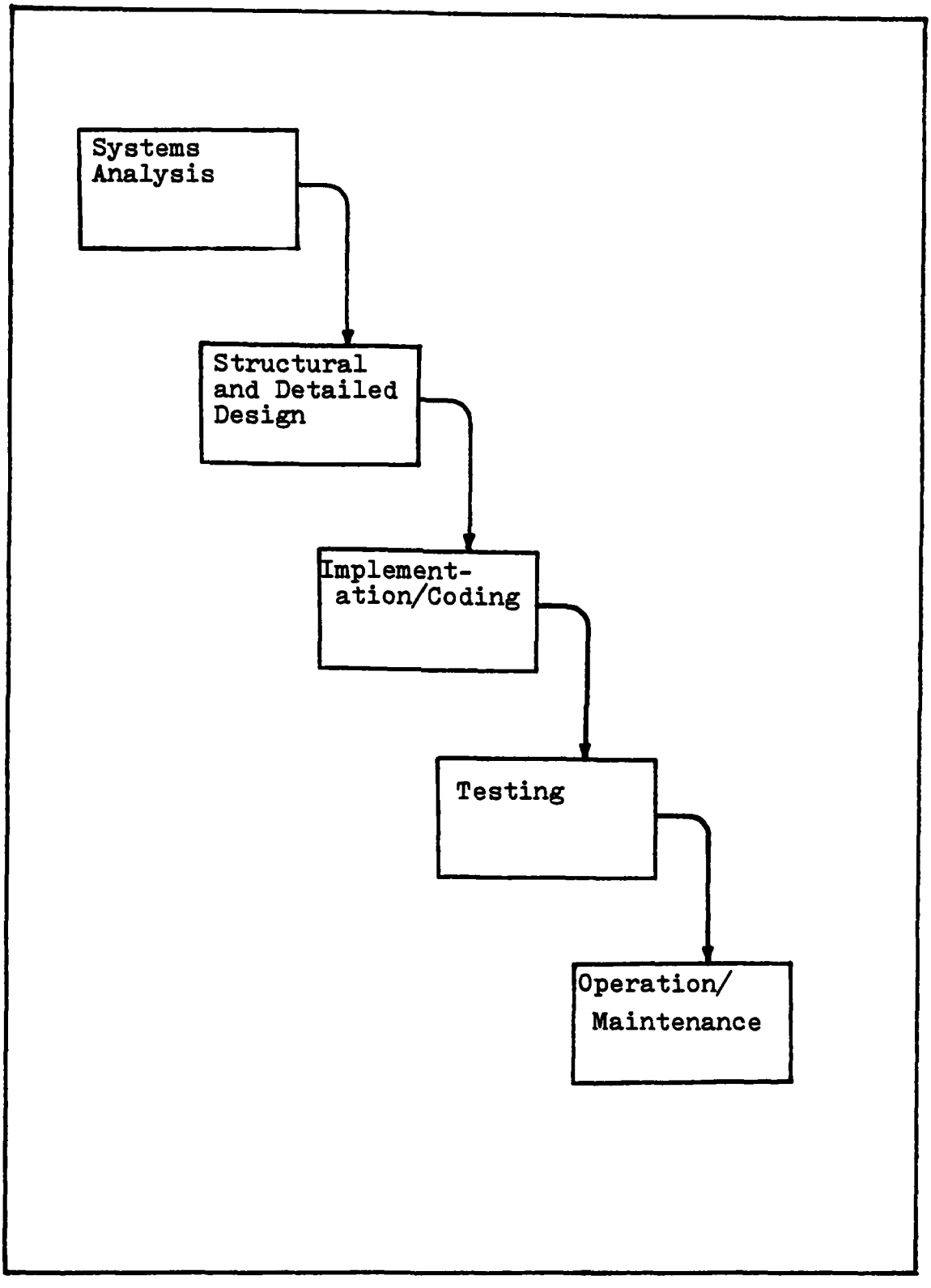


Figure II-1 Software Life Cycle-Waterfall

In this chapter, the requirements for this project's operating system will be explained with most of the specifications included. The requirements phase defines an acceptable solution to the problem. During the requirements phase, the designer must understand exactly what the user desires from the system. If the requirements are incorrect and an error is not detected until later in the design phase or coding phase, then the correction of the error can take more time and effort. This is why the requirements phase is one of the most valuable of the developmental phases of any software project, although all of them need to work together to produce the end result.

Local Requirements

Other than fulfilling the requirements for some students' project, this operating system can be used in various ways. It could be used to teach courses in the areas of software engineering, operating systems, computer architecture, and computer languages. This operating system could be used on different computers for different courses and could possibly provide more computer services that would be less costly than an additional minicomputer (for example, the VAX 11/780 on the first floor of Building 640). Of course, this would be conditional on

keeping the cost of the computer system under $10,000 (Ref. 5: 11).

Air Force Requirements

As was stated (Ref. 1: 10), the Microcomputer Technology Branch of the Air Force Data Systems Design Center at Gunter AFB, Alabama was created to supervise the production and acquisition of microcomputer software. This project can be used to serve two beneficial roles: 1) it can give insight to Air Force Acquisition personnel to correctly specify the required software for microcomputers (Ref. 1: 10) and 2) it can provide a fully documented operating system that can be developed into a useful tool which the Air Force, as well as AFIT, would be able to utilize. This documentation consists of structure charts, process definitions, data flows, source code comments, and a users' guide which are located in Appendices B, C, D, E, and F respectively.

Capabilities

The required capabilities for the operating system have already been developed (Ref. 1, 4, 5). They are used as the end goals for the completed and running operating system and are listed as follows:

1. Multiuser Support
2. "Friendly" User Interface
3. Communications between Users
4. Resource Management System
5. Meaningful Error Diagnostics and Recovery Procedures
6. Device and User Support

These required capabilities are also considered as the basic requirements for the design of the operating system. Each of these will be explained in the rest of this chapter.

Multiuser Support

One of the specifications that are given for this operating system is that it is to be multiprogramming. The definition given by Madnick and Donovan (Ref. 7) is "a term given to a system that may have several processes in 'states of execution' at the same time (a process can be in a state of execution and not be executing; that is, some intermediate results have been computed but the processor is not currently working on the process)" (Ref. 7: 7).

Multiprogramming is used in an environment that will handle concurrent users (as does the VAX 11/780 that was mentioned earlier). The first required capability, multiuser support, will require the design of a multiprogramming environment for the operating system.

For this operating system, the support of a minimum of four users has been given as the multiuser requirement for the implementation. The maximum number of concurrent users that can use the system will depend on three hardware constraints: 1) the number of serial and parallel I/O ports that can be used for Cathode-Ray Tube consoles (CRTs), 2) the memory constraints that are built into the operating system's data base, and 3) the size of main memory that is allotted for the users.

The design of the operating system will be for a multiuser environment of approximately eight concurrent users. The initial coding and implementation will be for four concurrent users and will be easily upgradable to five, and up to the maximum of eight, users. The upgrading process will be discussed in detail later. The initial implementation of four users will be an adequate test bed for the multiprogramming requirement by providing various CPU and I/O bound processes which would test out the software (Ref. 5: 11).

User Interface

The user interface has become an important aspect of any operating system. It is essential to provide a "friendly" user interface which can be utilized by users with differing computer skills and backgrounds. The user should be able to easily learn how to operate the system. The "friendly" user interface of any operating system is characterized by three qualities: 1) ease of use 2) tolerance of user errors and 3) minimization of user error opportunities (Ref. 6: 270-273). The operating system will work efficiently with an experienced user and still be able to assist a novice user in learning how to operate the system (Ref 5:12). Documentation for the user will greatly help facilitate the learning process.

Inter-User Communication

The inter-user communications of the operating system can be done using a mail routine, public files, or both. With a mail routine, one user would be able to set up a file to send to another user, while setting a mail flag informing the second user of mail. By using a public file system, any user can declare a file as public so that anyone can read, link, or list that file. The one restriction for this method is no alteration (erase,

overwrite, etc.) could be done to a public file, except by the original user. These two, mail routine and public files, can be combined in the system. The mail routine can be used exclusively for messages, and the public file system can be used for program files.

The mail routine and the public file system are solutions for the design. The requirements for the system's inter-user communications would be met by using one or both of them.

Memory and File Management

Memory and file management is concerned with four basic functions (Ref. 7:105) :

1. Keeping track of the status of each location of main memory.
2. Determining a policy for memory allocation.
3. Allocation technique.
4. Deallocation technique.

The status of each location of primary memory will be either allocated or unallocated. Allocated means that the memory location is being used for a job. Unallocated means that the space is free for any incoming jobs.

The policies for memory allocation will be influenced greatly by the following three constraints: 1) the maximum number of jobs allowed on the system at a time, 2) the desired turn-around time for average jobs (e.g. shared program modules), and 3) the size of jobs versus the size of main memory (i.e. can all jobs fit into the allotted working area). A few examples of memory allocation policies are partitioned memory, paged memory, demand-paged memory, and segmented memory (Ref. 7: 106). The latter two provide a virtual memory feature that will be discussed later.

Allocation and deallocation techniques depend upon the policy selected for memory management. If paged memory management is selected the allocation technique will have to place the entire job into main memory into a series of blocks. When the job is completed the job needs to be removed from main memory and the former allocated area will be returned into free. The programs that are active in memory need to have protection, which is accomplished by the allocation technique.

Error Handling and Recovery

The operating system should have the ability to handle and recover from system and user errors. Not only should it handle and recover from the errors, it should

also provide informative diagnostics that would help the user better understand the error. There are two types of user programming errors: fatal and non-fatal. These two types of user programming errors can be handled by displaying error messages (the diagnostics) and returning to the operating system's control. The user could also have format errors with the command language. This would simply be handled by giving the user the correct format for that particular command as the diagnostics.

The system can be modified for more users, for a different management routine, or for a different 16-bit microprocessor. There should be error detection capabilities for each of the new system modifications. This type of error detection would be useful for the person(s) trying to complete the modification.

Device Support

The device support will be handled by the Input/Output (I/O) Manager. These devices can be described as "the computer system's means of communicating with the world outside of primary memory. This communication may be with humans external to the computer system or with other parts of the system (such as tapes, computer cards, or disks) not directly accessible by most of the instructions of the central processor" (Ref. 10:

169). The four requirements for the I/O Manager (Ref. 5: 15) are the following:

1. Information transferral between users and I/O devices
2. Conversion of the user's view of I/O device (virtual I/O) to physical characteristics
3. Sharing of I/O drivers
4. I/O device error recovery

There are three major techniques used for managing and allocating I/O devices: 1) dedicated, 2) shared, and 3) virtual (Ref. 7: 284). The third requirement would be implemented using a shared devices technique and a virtual devices technique. The shared devices technique would allow such devices as disks, drums, and most other Direct Access Storage Device (DASD) to be shared concurrently by several processes (Ref. 5: 284). Slow I/O devices (such as teletypes, printers, and card readers) would have to use the virtual device technique (for example, a SPOOLing routine) in order to convert them into shared devices (Ref. 5: 285).

Design Approach

As stated in the first chapter, the top-down approach will be used for the most part in designing of the operating system. The break from the top-down methodology will be when an algorithm that is used in a lower level can be designed and written before that level has been reached.

Implementing Language

In the past, operating systems were written exclusively in assembly language (Ref. 8). The two main reasons are: 1) a well written and optimized assembly code is the fastest-executing code available and 2) the code produced using assembly code is the most compact, taking up the least amount of memory (Ref. 1: 13). With the cost of memory going down, the second reason is not as critical, since more memory can be purchased. With the use of structured designing (such as the top-down approach), the use of structured languages can be written to follow the physical form of the design used.

In the design of the operating system, the use of a high-level structured language will be used for the control structures, and the assembly language of the microprocessor will be used in those areas where

performance needs to be optimal (such as the device drivers). Two examples that have used this type of hybrid design are: 1) UNIX, written in the C language and 2) UCSD Pascal, written in Pascal (Ref. 1: 14).

One of the specifications for this project's operating system is portability. Since the bulk of the operating system will be written in the C language, it can be transferred and compiled easily. The assembly language routines will need to be rewritten in the new microprocessor's assembly language.

As stated in the first chapter, the C language was chosen as the high-level language that would be used in the final implementation, with the assembly language used for some of the machine dependent routines that cannot be handled efficiently using the C language.

The C language is readily available at AFIT. A cross-compiler from C to Z8000 assembly code is available on the VAX 11/780 located in the Digital Engineering Lab. A C compiler for the AMD Z8000 processor is being acquired from AMD. When this compiler is received, it will be used to compile the C portion of the operating system. A Pascal compiler is available but C was chosen because it is less restrictive (Ref 1: 15).

Microprocessor Consideration

The AMD Z8000 microprocessor was selected from an extensive study with three other 16-bit microprocessors (Intel 8086, Motorolla 68000, and DEC LSI-11/23). The LSI-11/23 was not considered for this project as a target device (Ref 1:16).

The amount of hardware support offered by the microprocessor was important in the selection of the target device. The following are the desired hardware supports that the Z8000 offers (Ref 1:17):

1. Restriction of CPU access.
2. Restriction of memory access.
3. Memory mapping and program relocation.
4. Sharing of memory.
5. Context switching support.
6. I/O interrupt support.

For CPU access restriction the Z8000 was the only one to differentiate between normal and system modes with restriction of the use of I/O instructions, control registers manipulation, and the HALT instruction. All the microprocessors require external circuitry to control access to memory, but the Z8000 provides instructions for use with the memory segmentation. When an interrupt is

received, the Z8000 has block move instructions for facilitating the storage of the entire instruction set, while the other microprocessors only store part of the machine state. The Z8000 allows the interrupt vector table to be located anywhere in memory, but all the microprocessors react in the same way to interrupts (Ref 1: 21).

Summary

The implementing language and the microprocessor were not requirements for the operating system. These two have been presented because they were previously selected for initial implementation (Ref. 1).

For any system to be designed correctly, the designer must completely understand what the user desires from the system. A successful completion of the requirement phase will enable the designer to provide the user with the necessary results. Since the operating system is a complex piece of software, the requirement phase is more important than in less complex pieces of software. This chapter presented the requirements for AMOS that were defined in previous thesis efforts.

The C programming language was previously selected (Ref 1: 22) for its clarity, power, and availability. For these same resons and for the availability of UNIX C

source code, C was selected for this effort.

The AMD Z8000 microprocessor was chosen for the initial implementation. The Z8000 enables the operating system to:

1. Easily handle main memory.
2. Differentiate between system and user tasks.
3. Efficiently handle interrupts.

The testing requirements for AMOS are module testing, system testing, and acceptance testing. Module testing includes the testing both the structure of individual modules and the integration between modules. System testing is the validation of the system to it's initial objectives (Ref. 25: 232). Acceptance testing is the validation of the system to it's user requirements (Ref. 25: 232).

III. Operating System's Design

Introduction

The purpose of this chapter is to present the detailed design of the AFIT Multiprogramming Operating System (AMOS).

The detailed design of this multiprogramming operating system was done using a top-down methodology. When using a top-down approach, a well structured design is possible. This top-down design approach is done by making the upper-level modules call only lower-level modules. This will allow easier modification and understanding. These are two requirements that were previously stated.

This operating system was designed with a structured programming language in mind, such as the C language which was selected. By using a well known and widely used programming language for operating systems, a portable system is obtained. The only modules that would need to be changed would be the low-level driver routines that will be written in assembly language. The need for this operating system to be portable was previously stated in the requirements chapter.

The tool used in the design of AMOS is the structure

chart. The structure chart was chosen because as the design is refined, new modules are identified and added to the chart (Ref. 11:60). The initial version of the structure chart is derived from the analysis tool called a data flow diagram (DFD) (Ref. 11: 61). The initial structure chart for AMOS was derived from the DFDs presented by Reference 4. The DFDs were not included in this text, because the structure charts can be presented apart from the DFDs and still be complete in its presentation of the operating systems structure.

Bootstrap

After the computer system is powered-on, the operating system should be loaded into main memory. The operating system is then executed. This is done using a bootstrap program (See Figure III-1). The Bootstrap does not meet any requirements, but is necessary for the operating system to be loaded and executed in a disk environment. Two ways that the bootstrap program can be executed are:

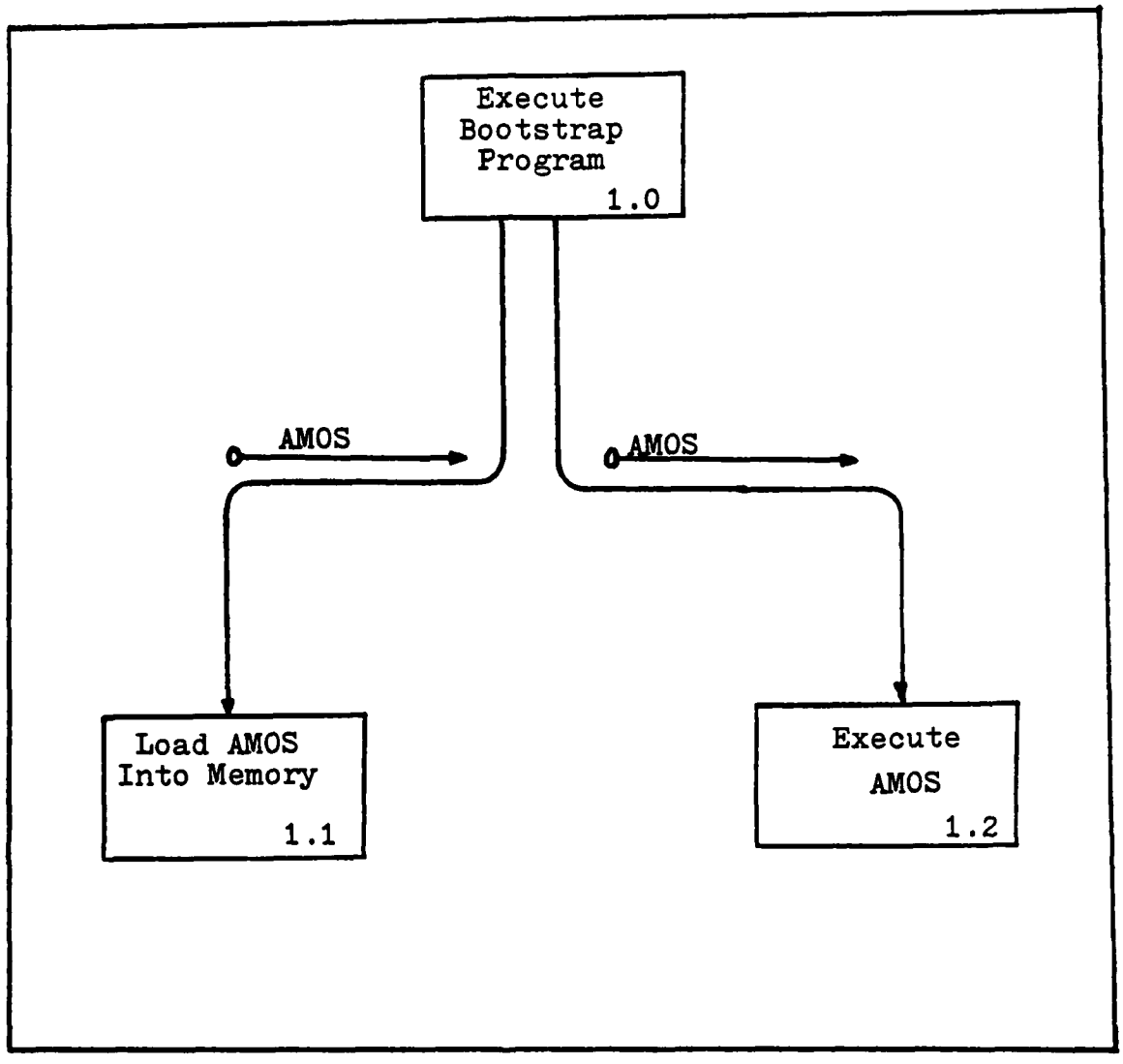


Figure III-1 Execute Bootstrap Program

1. The use of a low level function on the system's monitor. For example, on the TT-10 CP/M systems in the Digital Engineering Lab, the letter C which is entered by the user, initiates a monitor function that bootstraps the operating system. (Ref. 23)

2. The processor would be held in a RESET state while the disk controller independently loads a small segment of code from the disk. This code is the bootstrap program and is executed. (Ref. 1: 28)

Overall System Design

The system design was separated into the following three parts:

1. Intialization of Data Base.
2. Polling and parsing of the command.
3. Determination of the command type.

The initialization of the Data Base is to be performed only once during the execution of AMOS. The Data Base is the necessary information that the operating system requires in order to function properly. The polling and parsing of the command line and the determination of the command is in a infinite loop (See Figure III-2).

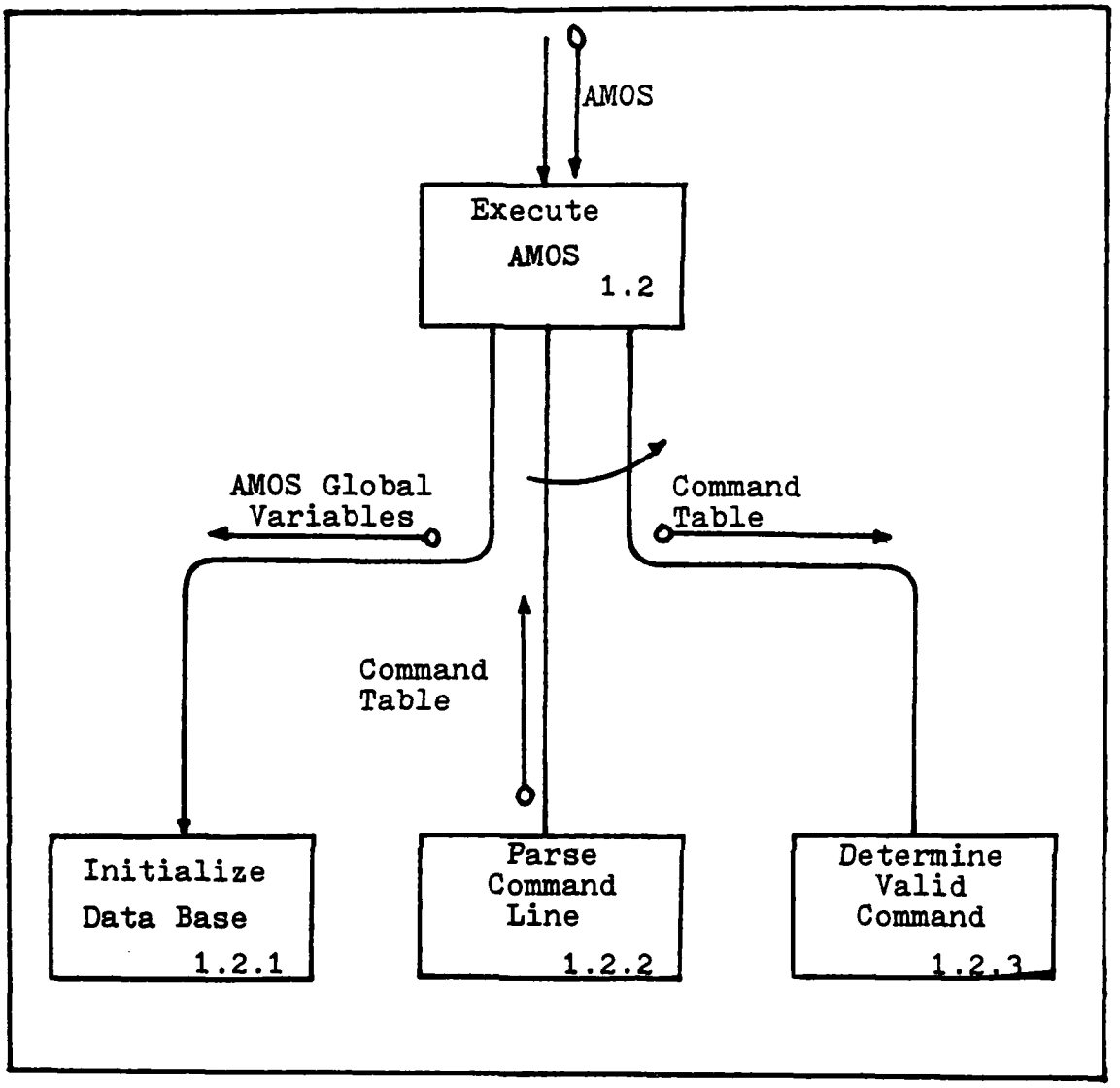


Figure III-2 Execute AMOS

Initialize Data Base

The operating system has its own set of data structures that it alone can access and control. For example, it needs structures that will give the status of processes (process control blocks or PCBs). When a computer system is powered-on, the main memory is not automatically cleared and may contain unwanted "garbage." When the data base is loaded into the main memory, it needs to be initially set to predetermined values (See Figure III-3). Some of these values can be changed during the operation of the operating system, while others can only be changed by in the source code of the operating system.

Polling and Parsing of Command Line

After the operating system is read into main memory, a polling routine is then executed checking the various ports for incoming commands. The polling routine will satisfy the user "friendly" environment. When a command line is received, it is parsed, or broken apart, into the various parameters. For example, for the RUN command, the command line is parsed into the command and file name. These various parameters make up the command table. This command table is used when checking the validity of

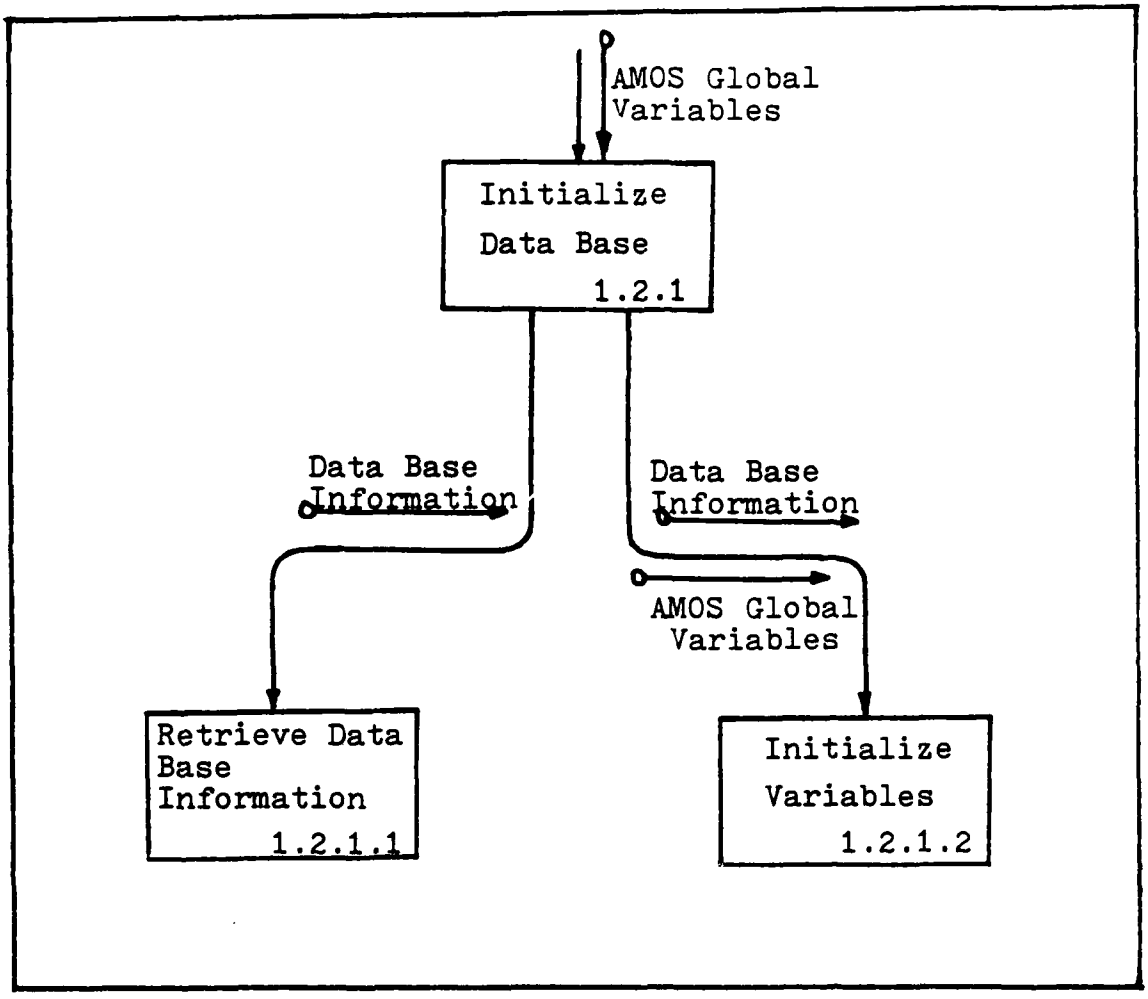


Figure III-3 Initialize Data Base

a requested command and in executing the command (See Figure III-4).

Determine Command Type

The command will be one of the following types:

1. User log-in.
2. User log-out.
3. Help user.
4. User commands.
5. Systems commands.

The first two command types are canned routines, which means execution of these are similar for each request. The latter three can vary for separate requests having different parameters.

The user is attempted to be logged-in. If the user is found to be already logged-on, then one of the remaining four command types is performed. If the command line does not follow the defined syntactical format, an error handling routine is called. After the error routine is completed, control is returned to the main body of the system (See Figure III-5).

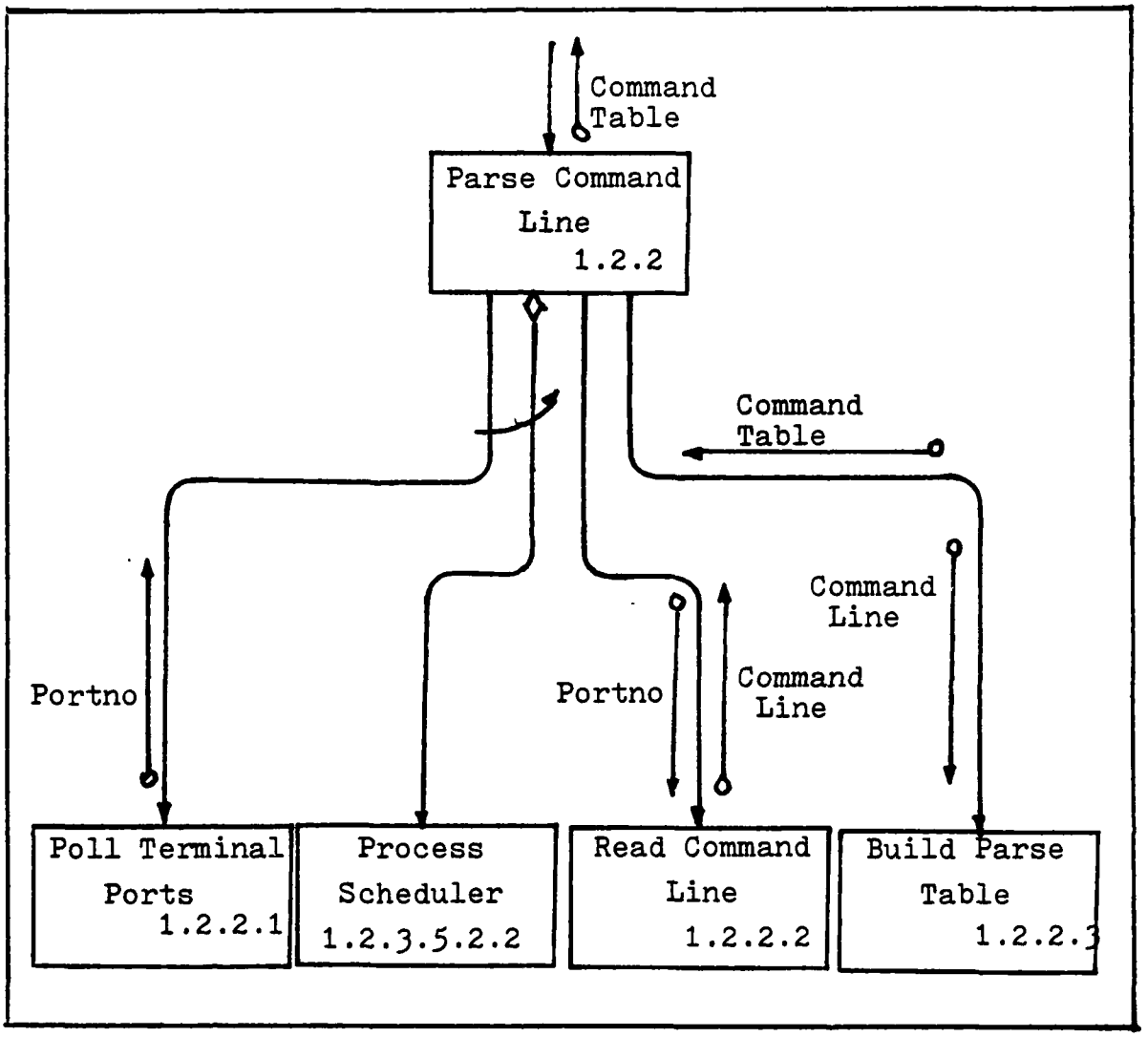


Figure III-4 Parse Command Line

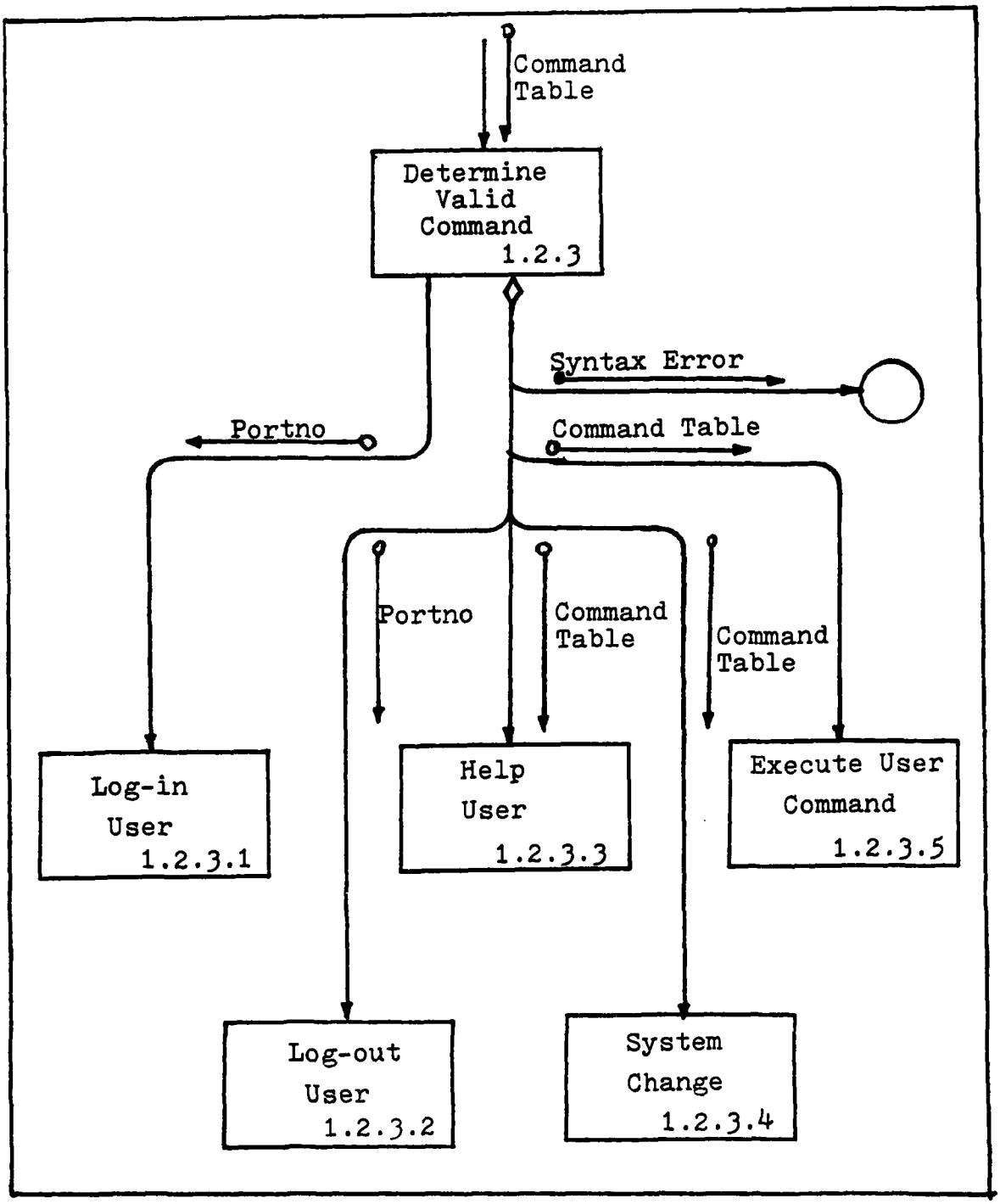


Figure III-5 Determine Valid Command

Note: (A) is a connector to Build Error Message Module

Validating Command

When determining the command type, user log-in, user log-out, and help user are validated. User and system commands have certain parameters that must be verified before execution.

The file name and username are the parameters that are verified for user commands. The requested file is checked to see if it is located in secondary memory (e.g. disk). If the requested file is located in secondary memory, then the user name for the file is checked against the user's username to verify that it is a legal owner of the file. This design will allow for public domain filing. Public domain filing would allow any file to be accessed by any user that is allowed on the system.

The validation of system commands is done by checking if the user requesting a system change is the 'Superuser'. The 'Superuser' is the user authorized to perform system changes. The 'Superuser's username and password should only known by those individuals authorized for system access.

Execution of Valid Command

This module will ensure the use of a multiprogramming environment. This is done by having the System, Log-in,

Log-out, and Help commands to be executed without waiting and by having the user commands wait in the process queues.

System Command

The system command is a special command that cannot be used by any user logged onto the system. It should only be executed by the 'Superuser.' This 'Superuser' has a designated username and password, as do other users. The difference between the 'Superuser' and other users is the username for the 'Superuser' is part of the original source code. The password for the 'Superuser' should only be known by those designated individuals that will have authority over the computer system. The password for the 'Superuser' should be changed frequently to avoid any tampering by individuals that determine the password code.

The design of the execution of the system command is shown in Figure III-6. The authority of the user is verified by checking to see if the user is the 'Superuser.' If the user is not the 'Superuser,' then a unauthorized user message is sent to the user. If validation is completed and the user is authorized to use the command, then the system is configured based on the command. To configure the system, a menu is sent to show the options the user has to choose, along with a prompt to

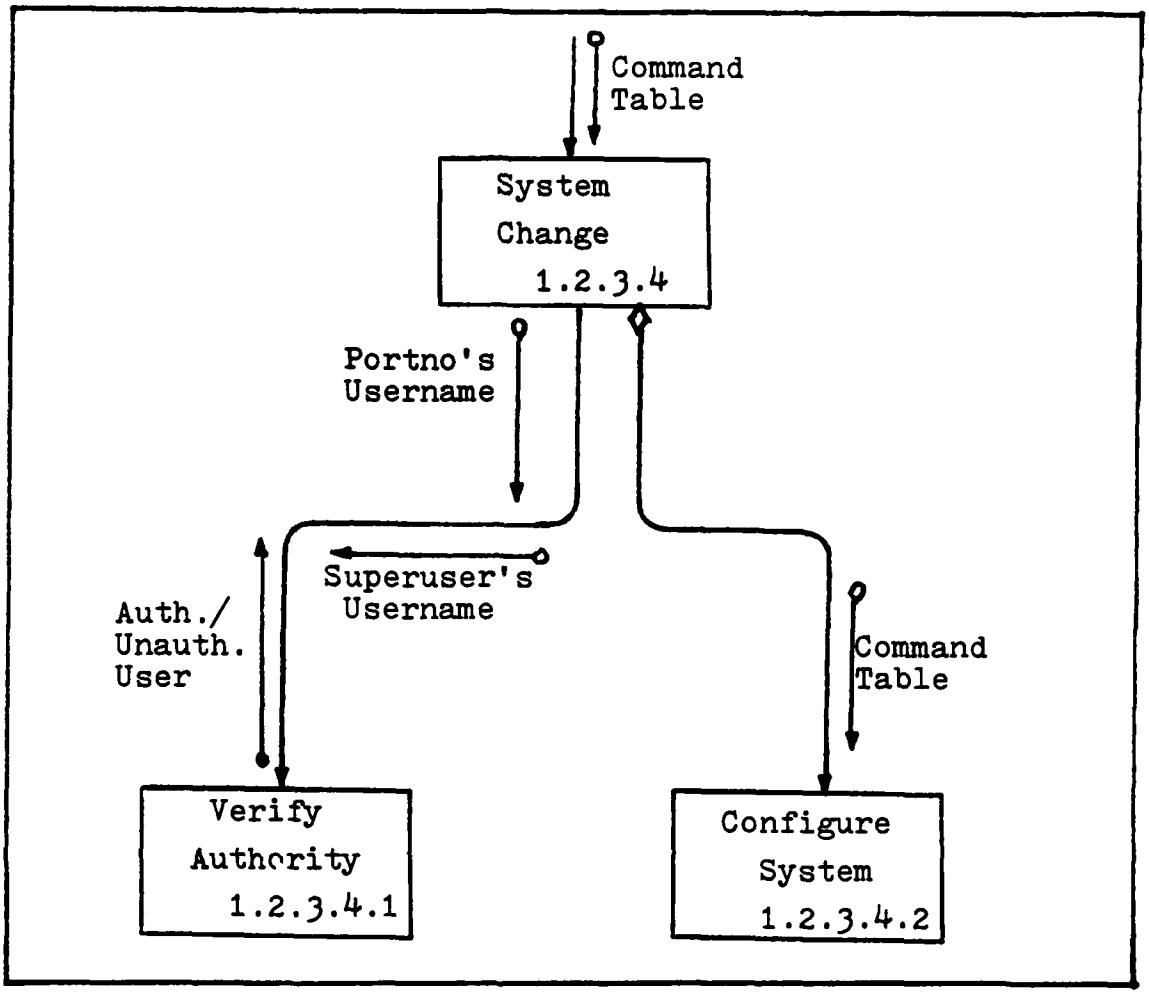


Figure III-6 System Change

input the information. When the required information is input by the user, the system changes are made and the system is configured to the new data.

Log-in User

Before the user can do anything with an operating system that uses usernames and passwords, the user must log onto the system. In the design, to log-in a user the following steps are to be taken (See Figure III-7):

1. The user inputs username and password when prompted.
2. The username is checked against the list of usernames that can access the system. If it is not in the list, then the user is not logged into the system. If it is on the list, the password associated with the username in the list is checked against the password input by the user. If it is not the same password, then the user is not logged into the system.
3. The user parameters are initialized to log-in the user and a logged-in message is sent to the user.

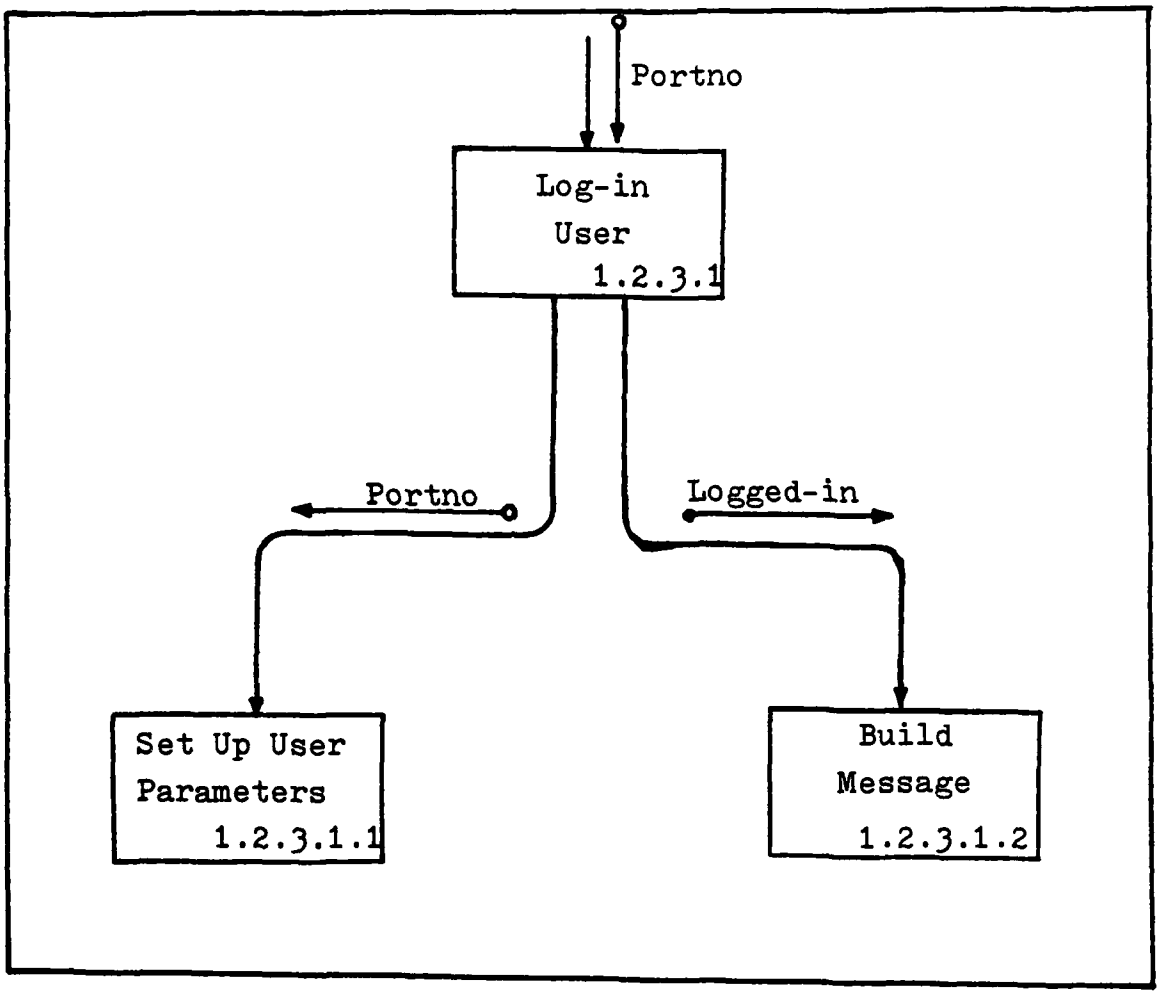


Figure III-7 Log-in User

Log-out User

An importance of logging-off of a system is the freeing of a terminal which can be used by another user. Logging-off tells the system that you do not need to do any more operations and are freeing the allocated workspace, terminal, and any other devices. In the design of the execution of the log-out command, the user parameters are cleared and a log-out message is sent to the user. The user parameters are the structures that inform the system about the user's current status. When the parameters are cleared, this informs the system that the user is no longer logged-on. The user would have to log-in to access the system, after the log-out command is completed (See Figure III-8).

Help Command

If a help command is received the user can be requesting either system information or command information. Whichever is requested the user is provided with the necessary information. The design does not include the necessary information for user log-in. This information will be documented and available for each user. This implies that a user must be logged in to request help.

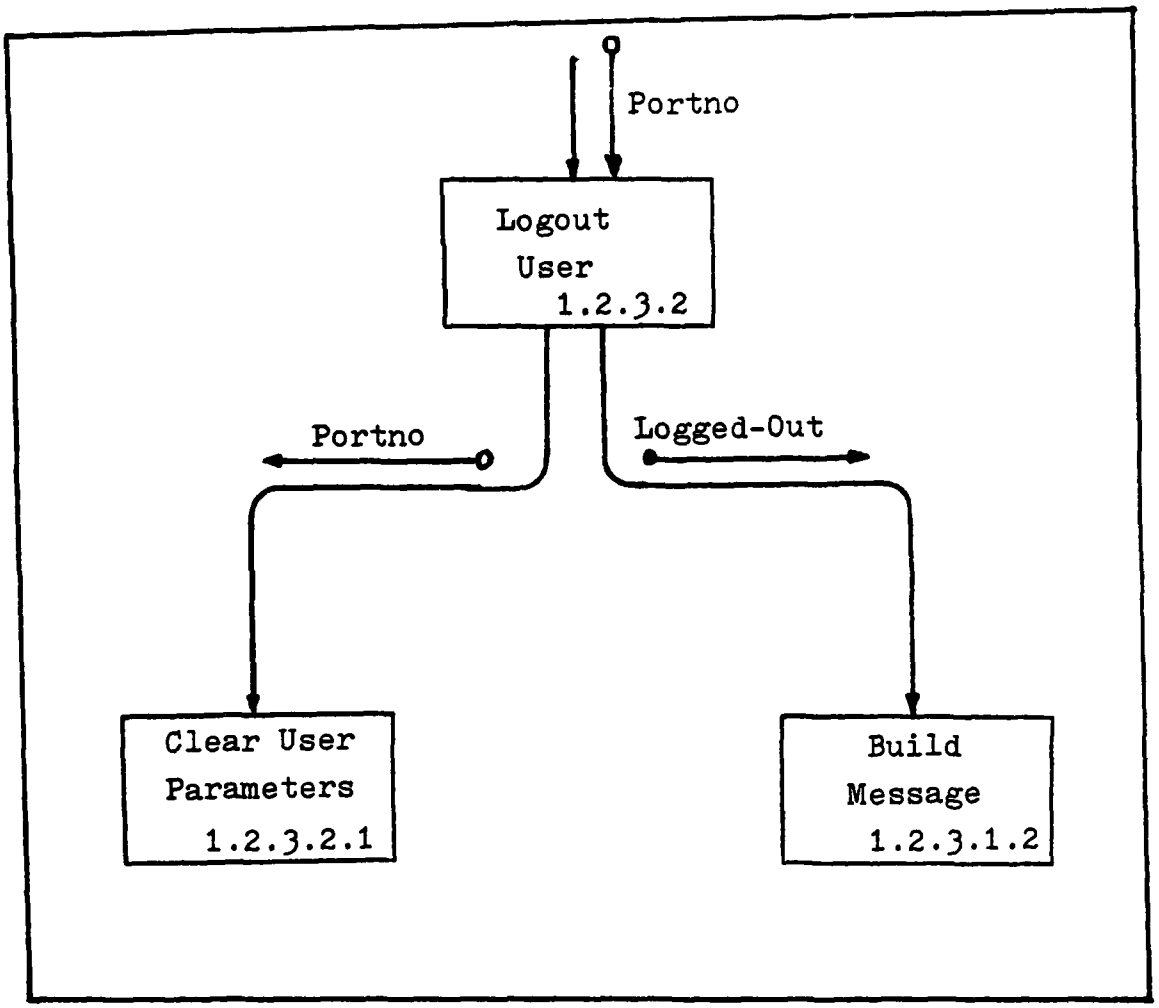


Figure III-8 Logout User

In the design of the execution of the help command, the information is retrieved and sent. Before the information is retrieved, the command is checked to determine if the user is requesting command information or system information. Command information is the format of commands and the description of what the commands do. System information is the status of the computer system (See Figure III-9).

User Commands

User commands are divided into the following five types:

1. Running a file.
2. Listing a file.
3. Printing a file.
4. Deleting a file.
5. Directory information.

The execution of all user commands are similar. First the file must be located (the directory is saved in a specific location), second the file must be retrieved from secondary memory, third the file is buffered into main memory, fourth the job is placed into a waiting queue for execution. After the four steps the files waits for

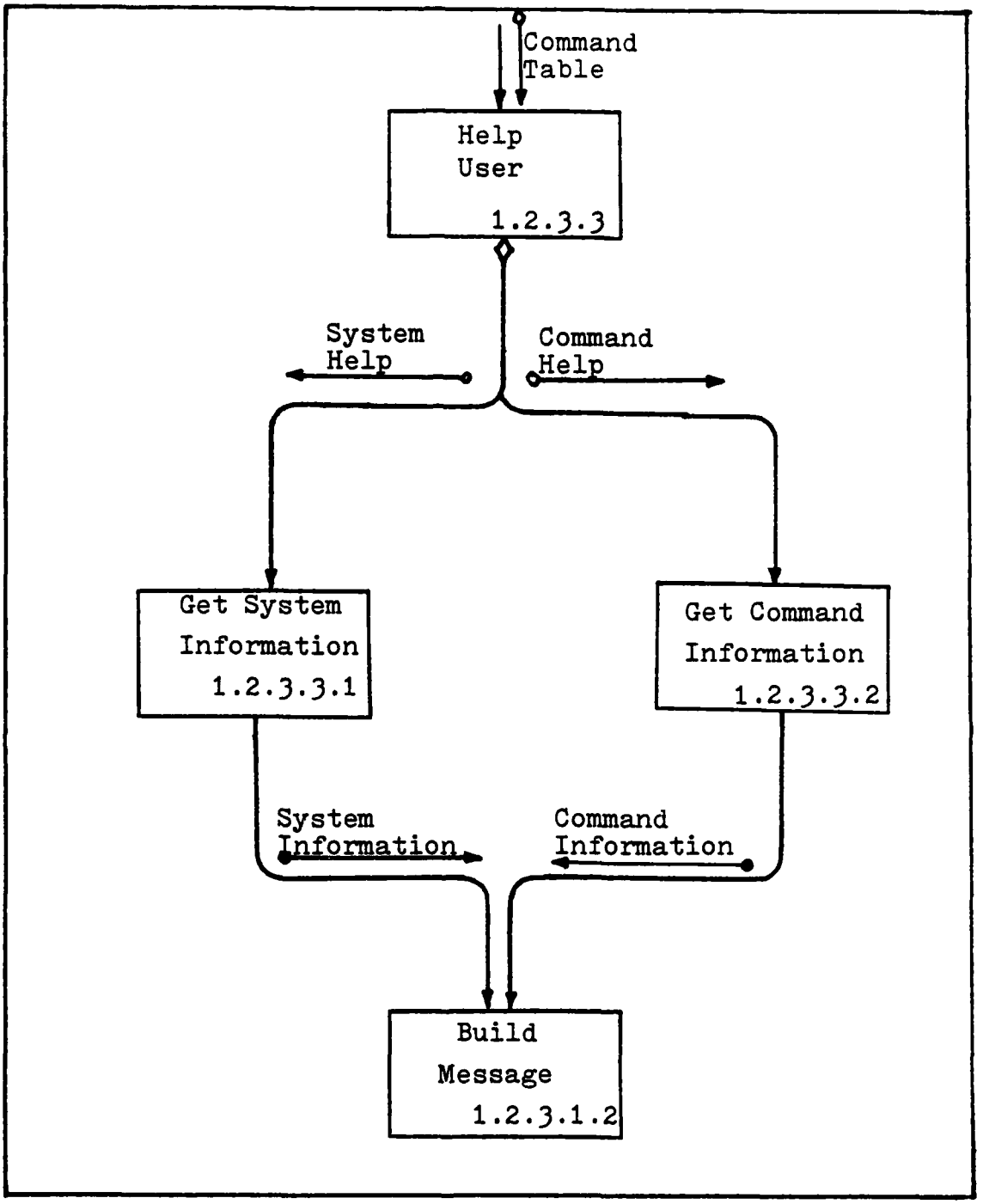


Figure III-9 Help User

its turn for execution.

To run a file, the executable code is located in main memory and then executed for the user. To list a file the file is taken from main memory and transmitted to the user's terminal. Printing a file is similar to listing a file except it is transmitted to a printer. Deletion of a file is done by buffering in the directory section containing the file information and deleting it from the listing and then writing it back out to secondary memory. Directory information is executed by transmitting the buffered directory to the user's terminal (See Figure III-10).

Evaluation of AMOS Design (Testing)

The evaluation of AMOS design consisted of a two phase process. The first phase checked to see if the design had a logical structure. A logical structure should have upper-level modules calling lower-level modules in a organized sequence. The sequence would follow a logical flow. For example, it is necessary to call a module getting a username before calling a module to check if the username is valid. The second phase determined if the data flow between modules was logical. The data would be needed in the called module or in another lower-level module. This lower-level module is

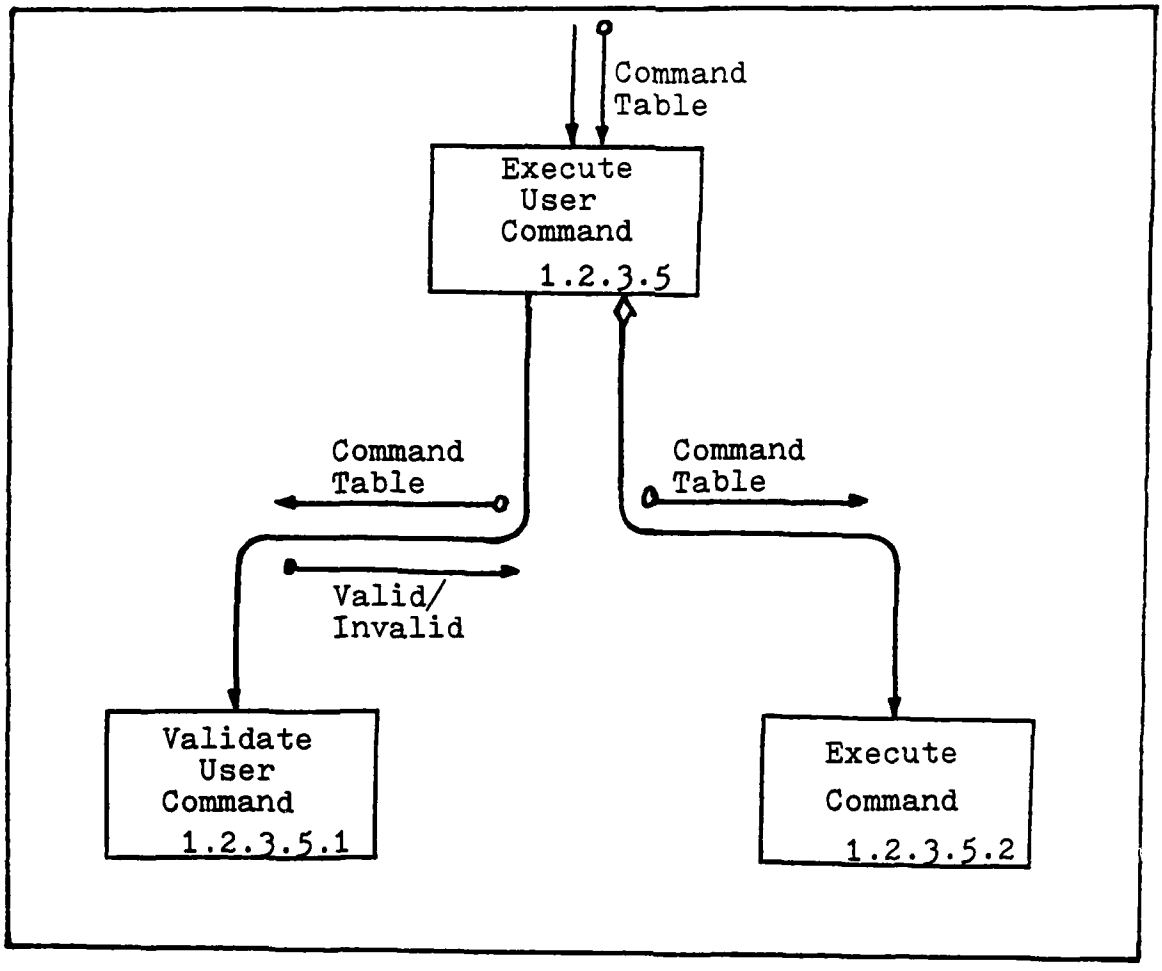


Figure III-10 Execute User Command

a child of the called module (Ref. 13: 220).

In the evaluation of the logical structure, a few minor changes were necessary. One of these changes involved the polling routine. The first design did not take into account that the users could be idle for a long period of time while a process was ready to run. The call to the Process Scheduler was added after a set number of polls with no response. Another change involved execution of a user command. The execution of each user command was similar; this allowed for common modules that would be called to execute the commands.

The data flow was verified to be logical with a few changes. To execute a command, in the initial design, the command table was the data flow item passed between the modules. When the Process Scheduler was incorporated into the design, the process control block was the new data flow item.

Summary

This chapter presented the overall system design of AMOS. The top-down design approach was followed using structure charting. The operating system was designed with the idea that it would be implemented using a structured programming language, i.e. C.

The overall system design is separated into the following three parts:

1. Initialize of Data Base.
2. Polling and parsing of the command.
3. Determination of the command type.

These three parts are the first level of the structure chart and call other modules in lower levels.

The structure charts are located in Appendix B. The process definitions and data dictionary are located in Appendices C and D, respectively.

IV. AMOS Process Management

Introduction

In this chapter, the AMOS Process Management genaral and detailed design is discussed. Process management is composed of the job scheduler, processor scheduler, and traffic controller (Ref. 7: 209-211). The job scheduler is the "overall supervisor" that keeps track of jobs and creates corresponding processes (Ref. 7: 212). The process scheduler performs the following four functions (Ref. 7: 213):

1. Keeps track of the status of the process
2. Decides the time the process gets to use the processor
3. Allocates the processor to the process
4. Deallocates the processor

The AMOS Process Manager will provide the desired multiprogramming environment. It will also help facilitate the "friendly" user interface by not making a user's process to wait for another process to finish an I/O function.

State Model and Structure Charts

The path of the process can be shown using a state model (Ref. 7: 211). Figure IV-1 shows the path that will be taken by a process in the AMOS Scheduler. The structure charts are derived from the Data Flow Diagrams (DFDs) that are presented in Reference 5. As was stated in Chapter 3, the DFDs are not presented because the structure charts show the structure by themselves.

Scheduling Techniques

There are two basic types of scheduling techniques: nonpreemptive and preemptive. In nonpreemptive schedules, a process once started is executed until completion (Ref. 12: 51). In preemptive schedules, it is possible to suspend the process from execution and at a later stage resume execution at the point of suspension (Ref. 12: 51). The combining of these two results in what is called selective preemption (Ref. 2: 199).

The nonpreemptive scheduling was chosen for the initial implementation of AMOS because of the simpler algorithm. In this simpler case the need to save the state of the preempted process is eliminated for other than interrupts. In the nonpreemptive scheduling, the process scheduler would give an indefinite amount of time

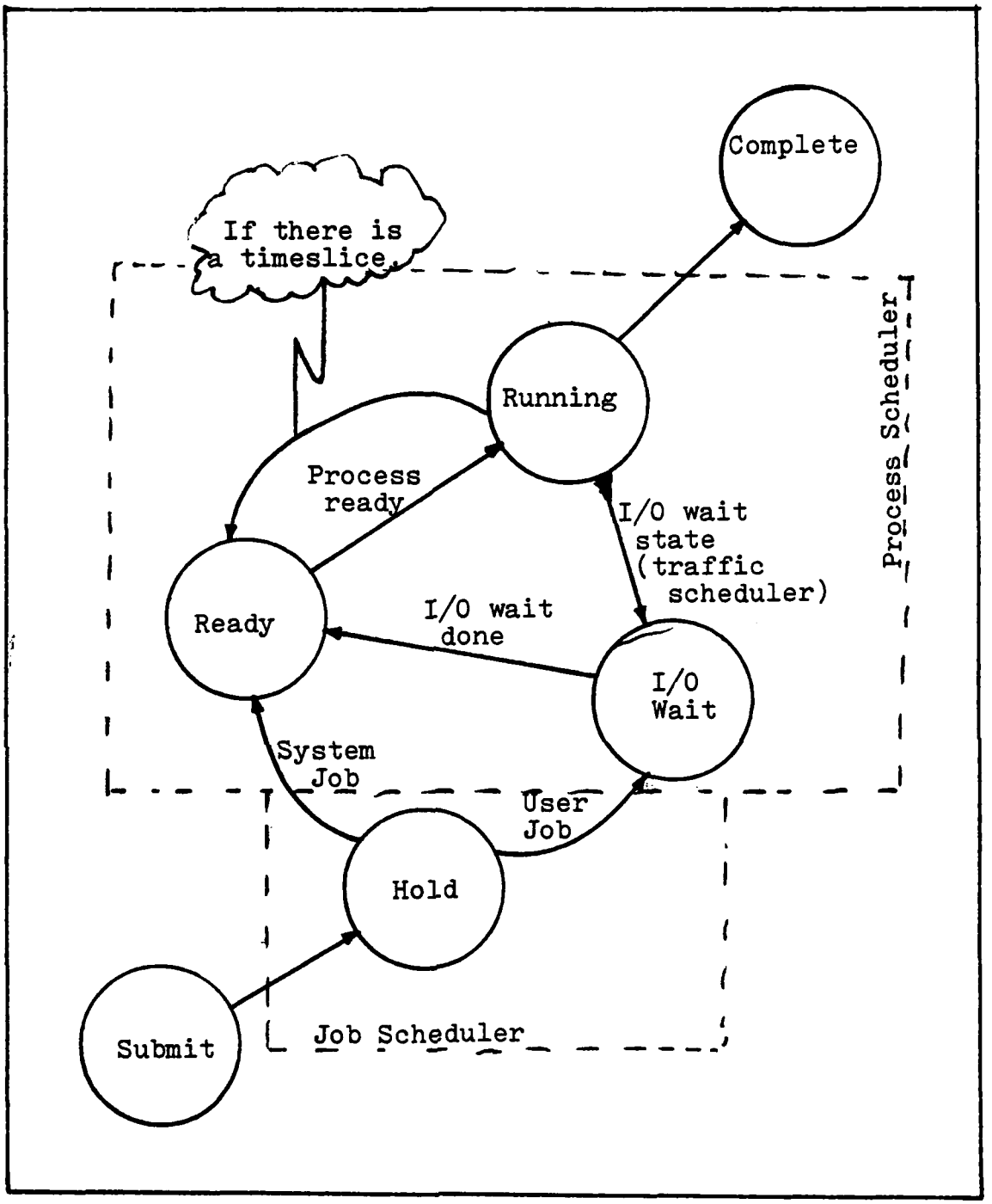


Figure IV-1 State Diagram

to the process, allocate the processor, and deallocate the processor after the process is finished.

After the addition of more terminals to the computer system, the scheduling could be changed to some kind of preemptive scheduling. If the waiting time becomes to lengthy for the users to get their jobs executed, timeslicing of processing should be implemented. The scheduler that was designed by Yusko (Ref. 5) has the timeslicing scheduler.

Job Scheduling

Since the nonpreemptive scheduling is being used, the job and process were defined to be the same. This allows for one table to be used for each process without the need to "cleanup" other tables that would only be used during a portion of the processor's execution cycle. That is, a users job will be treated as one process and not broken up into many processes. This is done by only having the User commands passed through the Scheduler. The other commands; Control, System, and Help; are "dedicated" and are not passed through the Scheduler. The term dedicated means that the job is run without anything interrupting it. This will be detailed in the design.

When a process is entered into AMOS, two procedures have to take place before it can be executed. The process

control block (pcb) must be constructed and initialized for the process. After the pcb is constructed and initialized, it must be inserted into a queue. The design of the 'job scheduler' can be seen in Figure IV-2. The 'job scheduler' is the module Build Pcb.

Process Control Block (PCB)

The pcb is the data structure containing the information describing the process (Ref. 3: 167). The following is the information in the pcb that is used by the process scheduler:

1. Priority of the process
2. Current Queue the process is residing
3. Offset Address, beginnin address of main memory where the process resides
4. Final Address of main memory where the process resides
5. Command type of the process
6. Process's port of origin
7. Process's I/O wait status

The insertion of the pcb into a queue also uses the information. If the pcb is waiting for I/O, then the pcb is inserted into the I/O Wait queue. If the command type

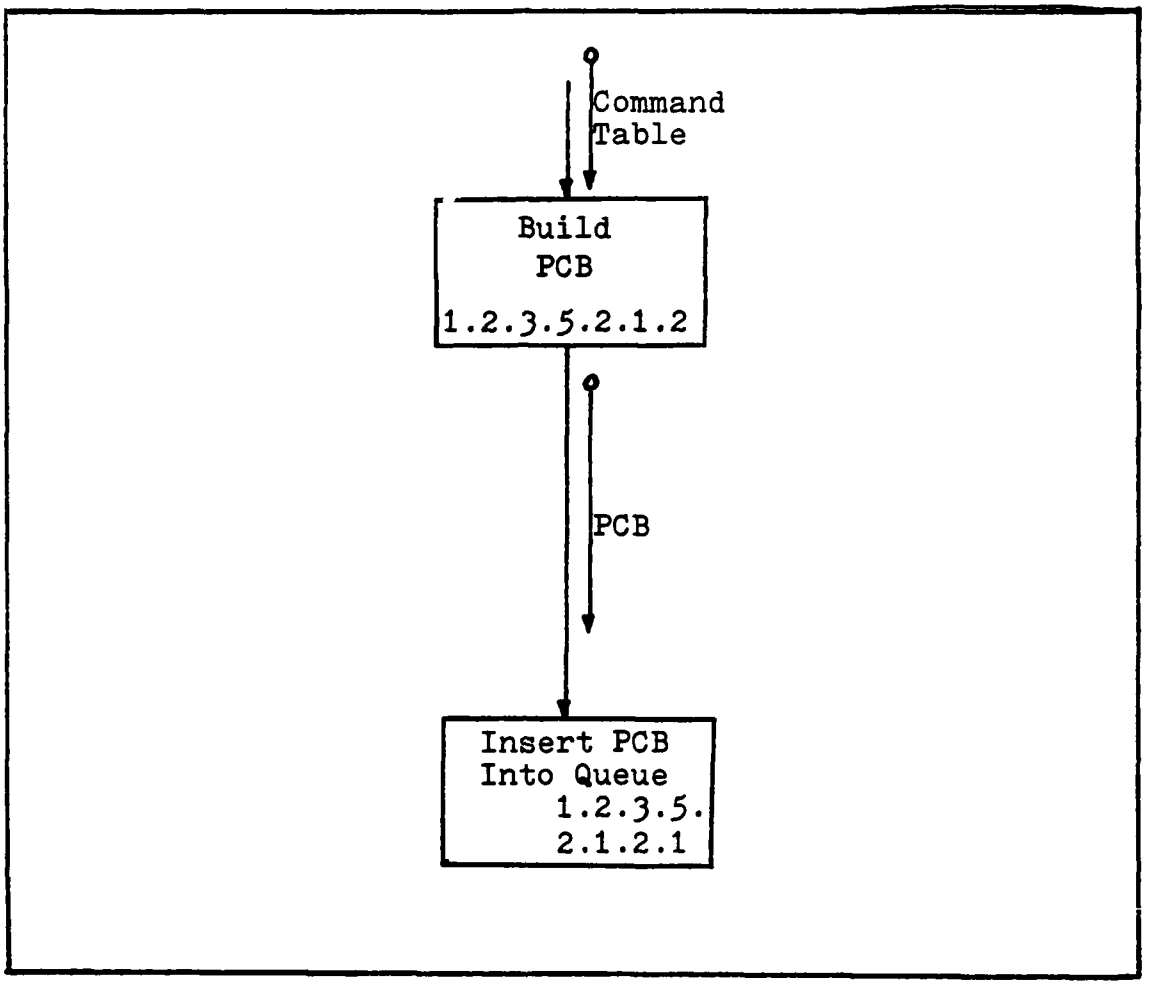


Figure IV-2 Build PCB

of the pcb indicates a system command, then the pcb is inserted into the System queue. If it was neither an I/O command nor a System command, then the pcb is inserted into a Ready queue. The Ready queue, which the pcb would be inserted, would depend on the priority of the process. The priority of the process is determined by the job scheduler and is given to the process when the job is entered and the pcb is constructed.

PCB Queues

All of the queues that are used by the Scheduler are dynamic first-in, first-out (FIFO) structures, because a dynamic FIFO structure is more flexible than a fixed size table. The queues, in the form of a dynamic FIFO structure, can vary in size and will contain only the space needed for the current pcbs occupying. When a pcb is inserted into one of the queues, it is the last one in the queue. This insures that a process does not get shuffled around in a queue with the possibility of not running until all other processes are finished. This insures that the user will have the process start execution before processes entered after the user's process was entered, assuming there is no I/O wait.

As was mentioned previously, there are three types of queues in the AMOS design. The three types are:

1. I/O Wait Queue
2. System Queue
3. Ready Queues

The I/O Wait queue is used to keep pcb's of processes that are in an I/O wait. An I/O wait occurs when the process tries to access a device and has to wait for a response. The System queue is used to store the pcb's of those processes that are system command processes. The Ready queues are used to keep pcb's of processes that are ready to use the processor and are waiting to be scheduled to execute. The number of Ready queues is determined by the number of priorities the system assigns to jobs. In the initial implementation, only one Ready queue (Ready1Q) will be defined. Since only four users will be on the initial implementation, it was not necessary for more than one ready queue, besides the system ready queue. If the need arises to start assigning priority base on needs of the processes in later implementations, it would be easy to add the extra Ready queues and change the code to account for the extra queue(s).

Design of Process Scheduler

The process scheduler was designed with simplicity in

mind, while still covering the four functions mentioned in the beginning of this chapter. The simplicity of the scheduler ensures that a multiprogramming environment can be implemented with the operating system without having to develope a complex scheduler. When the operating system is executing with the initial scheduler, a more complex, and more efficient in the use of the microprocessor, scheduler can be developed. In the AMOS process scheduler design (See Figure IV-3), the following modules are performed:

1. Updating all processes no longer in an I/O wait state
2. Get next process to execute
3. Execute the process

Updating of I/O Wait Processes

To update the I/O wait processes, the I/O Wait queue must be checked to see if there are any pcbs that are no longer in an I/O wait state. If such a pcb is found it is deleted from the I/O Wait queue and placed into the appropriate queue, either the System queue or one of the Ready queues (See Figure IV-4).

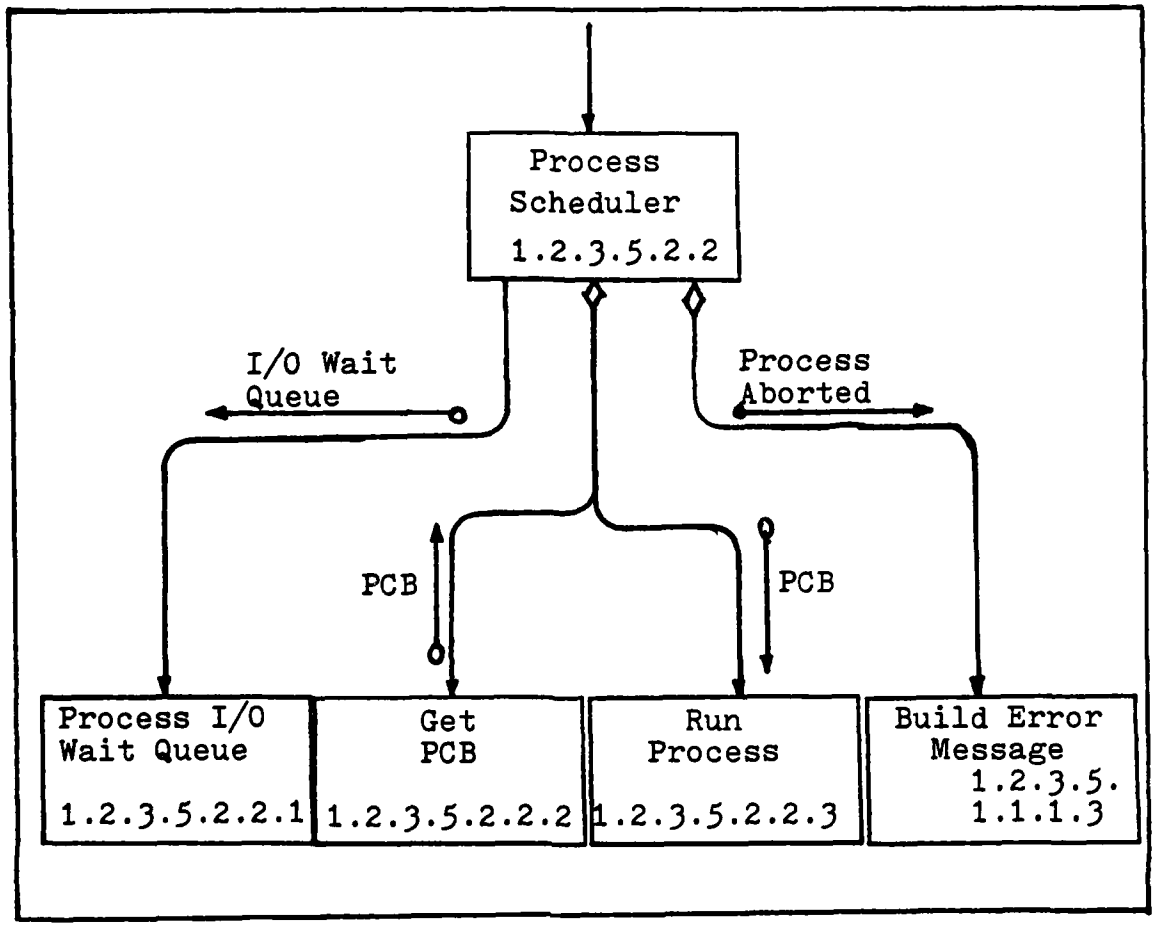


Figure IV-3 Process Scheduler

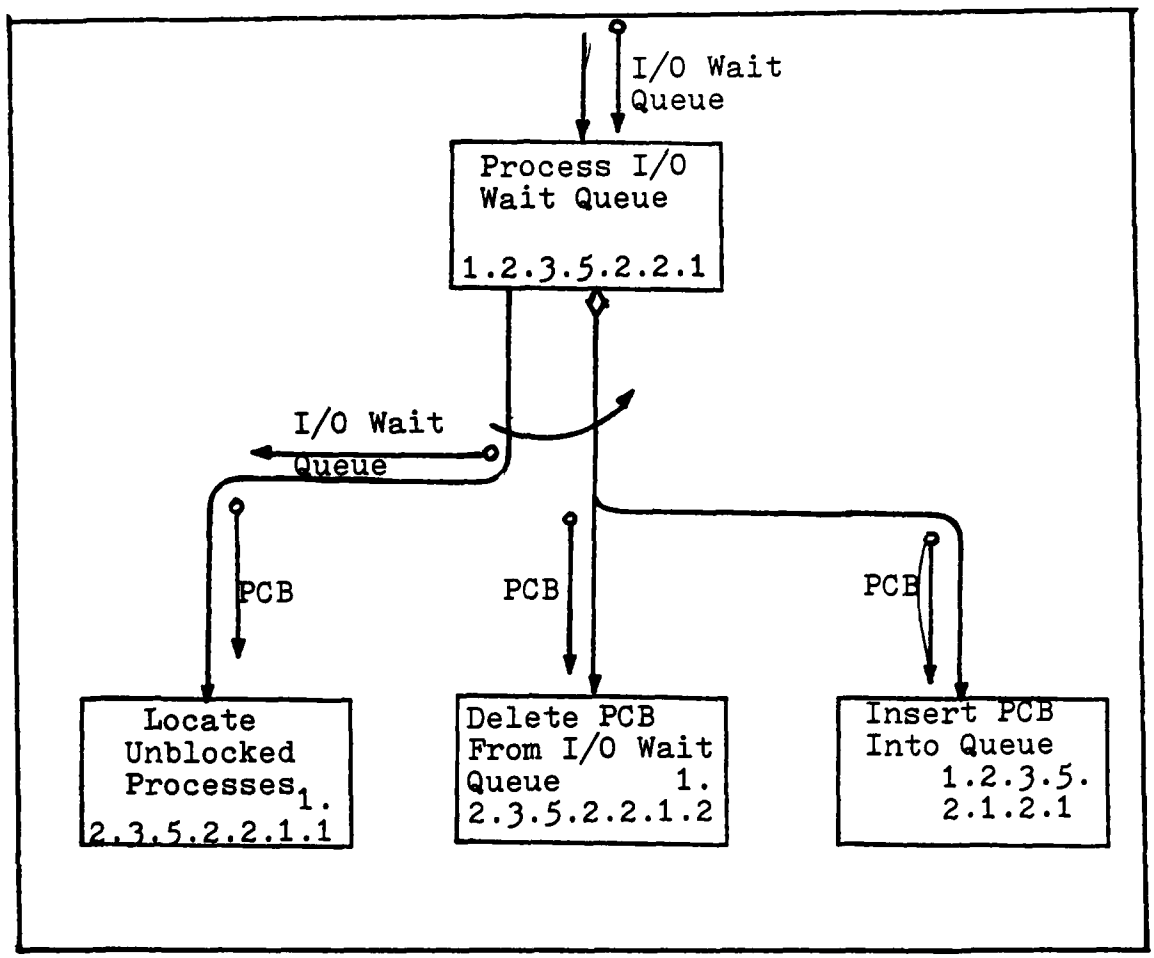


Figure IV-4 Process I/O Wait Queue

Get Next Process to Execute

When the Scheduler is ready to execute a process, a process that is ready to execute must be obtained from a queue. The process is taken from either the System queue or one of the Ready queues. The system processes have priority over the user processes, so the System queue is checked for a ready process and the Ready queues are checked only if there is no process on the System queue. A process' pcb is taken from either of the queues. If there is no ready processes, no pcb is retrieved and an empty flag is sent to the Scheduler (See Figure IV-5).

Execute the Process

If a ready process is found, then the Scheduler must decide what type of command the process originated (that is, User or System). This is accomplished in the Run Process module. If it was a SYS command, LIST command, PRINT command, DEL command, or DIR command, the process will be executed by calling the appropriate System command modules. If the process originated from the RUN command the module Run Program is called to execute the program. After the process is completed, or aborted, the module Deallocate Memory Space is called to deallocate the memory partition that was used by the process (See Figure IV-6).

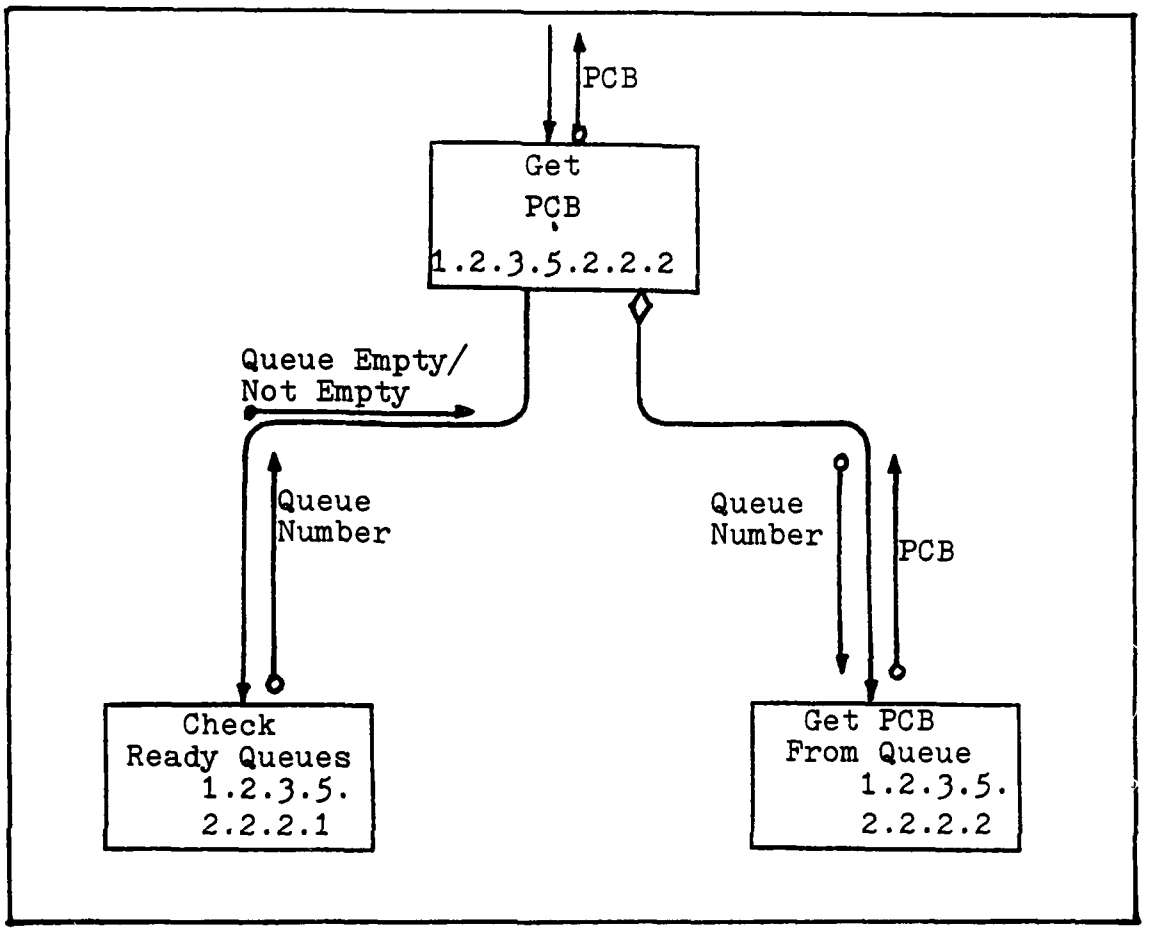


Figure IV-5 Get PCB

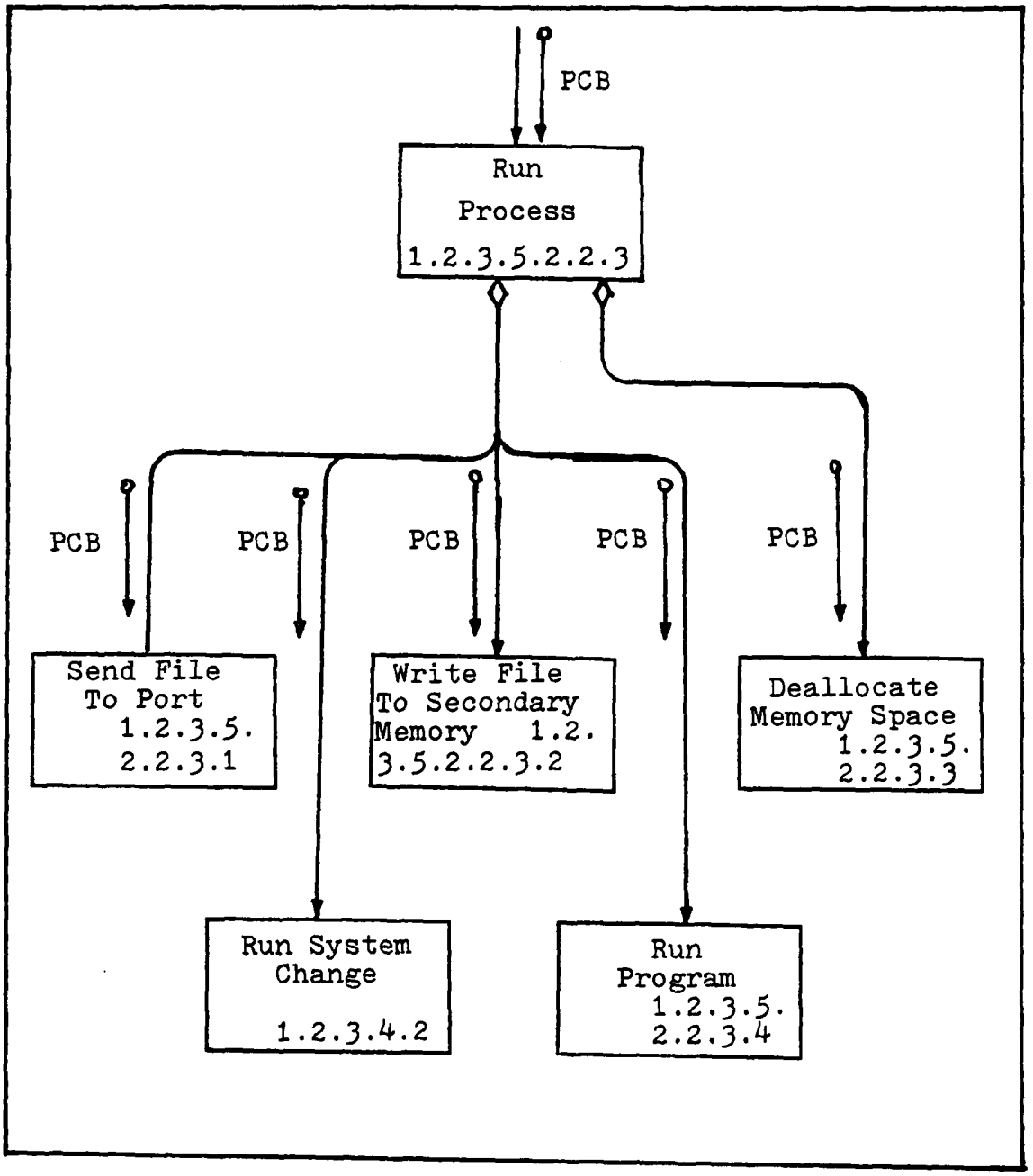


Figure IV-6 Run Process

In the initial implementation of AMOS, it was decided to allow only one process to be run at a time by a user. This is similar to the UNIX O/S running on the VAX 11/780, except that a process can be run in a 'background' mode (Ref. 8: 1918). The user can submit a job, but cannot communicate with AMOS until the job is completed or aborted. The multiprogramming environment will still exist, but only if there is more than one user on the system. Allowing for multiple jobs for a user on the system would be a job for the operating system's process manager to keep track of the processes.

AMOS Scheduler's Overall Design

Although the scheduler was designed in top-down fashion, the Scheduler, as a whole, was designed in two seperate parts with each part designed top-down. The two parts are the job scheduler and the process scheduler. The traffic controller is built into both of these and is not seperate from either one. The job scheduler is called from two seperate locations in the AMOS design, System Change and Get File (See Figures IV-7 and IV-8).

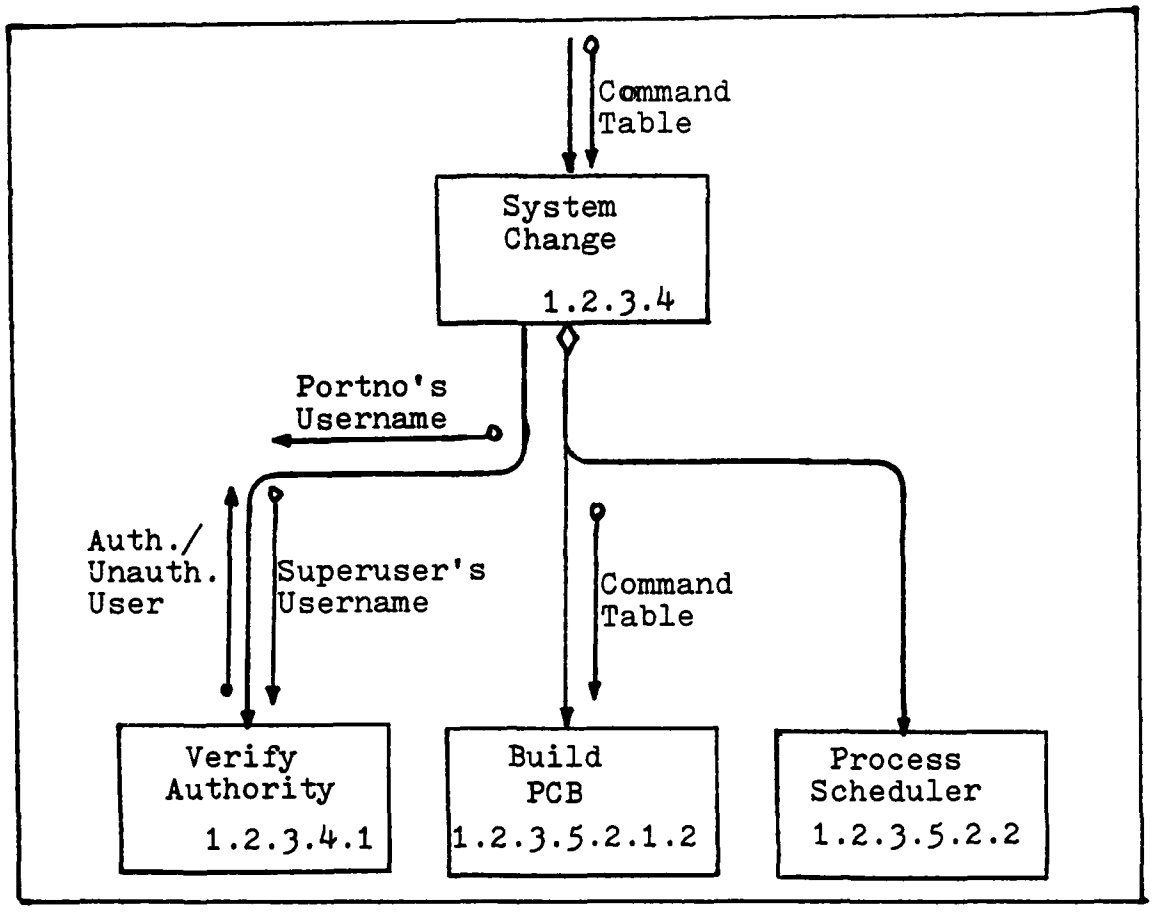


Figure IV-7 System Change

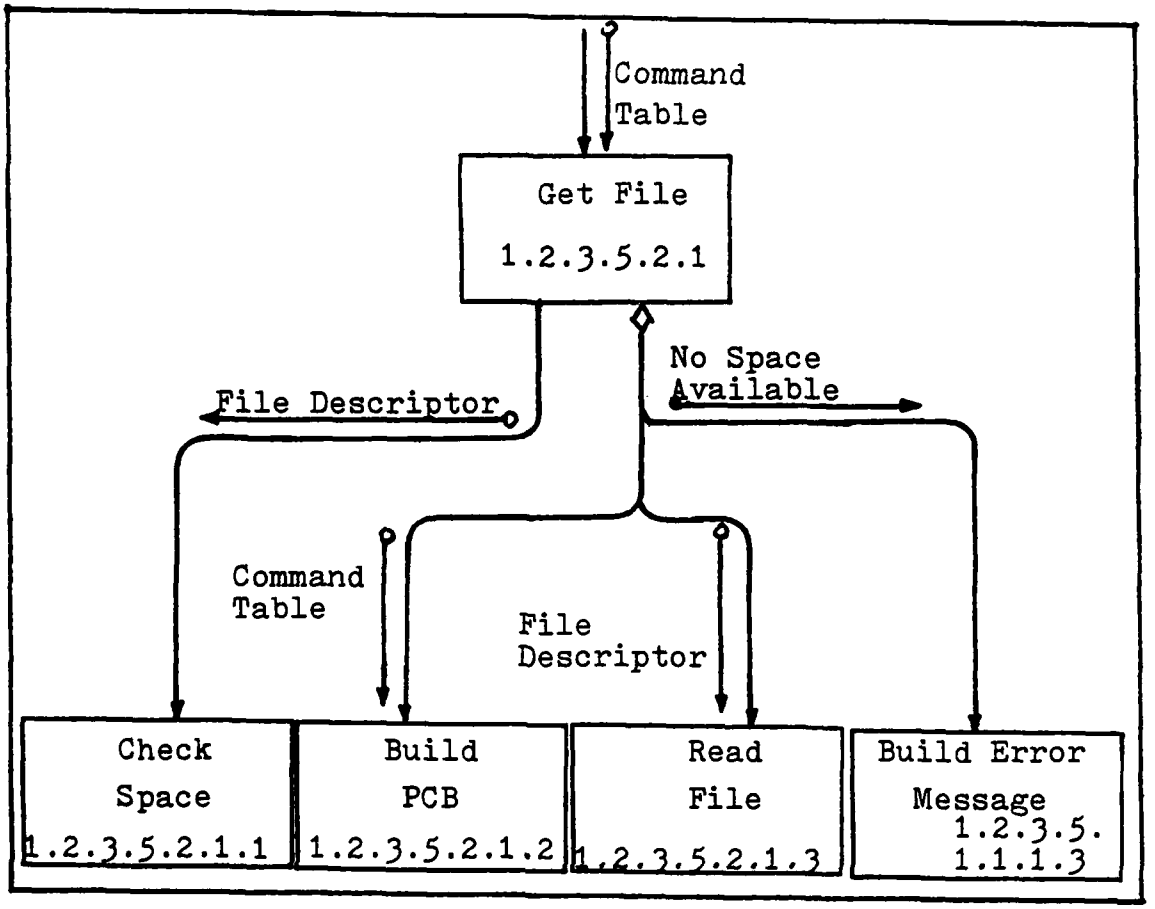


Figure IV-8 Get File

The process scheduler is called from two seperate locations in the initial implementation, Execute Command and Parse Command Line (See Figures IV-9 and IV-10), but can be called from a modified System change module (See Figure IV-7).

AMOS Scheduler's Implementation

Job Scheduler

The job scheduler was implemented using a simple algorithm. This algorithm is not in any textbooks, but was developed for AMOS. This algorithm uses the single table, the pcb, that was mentioned earlier. The algorithm is not as complicated as other job schedulers. Job schedulers that could have been implemented are a job scheduler using shortest job first and a job scheduler with future knowledge (Ref. 11: 218-220). The algorithm goes through two simple steps:

1. Building a PCB
2. Inserting a PCB into a Queue

Building a pcb consists of allocating and initializing. The allocation is a performed very simply. The pcbs are all declared in the beginning of the source

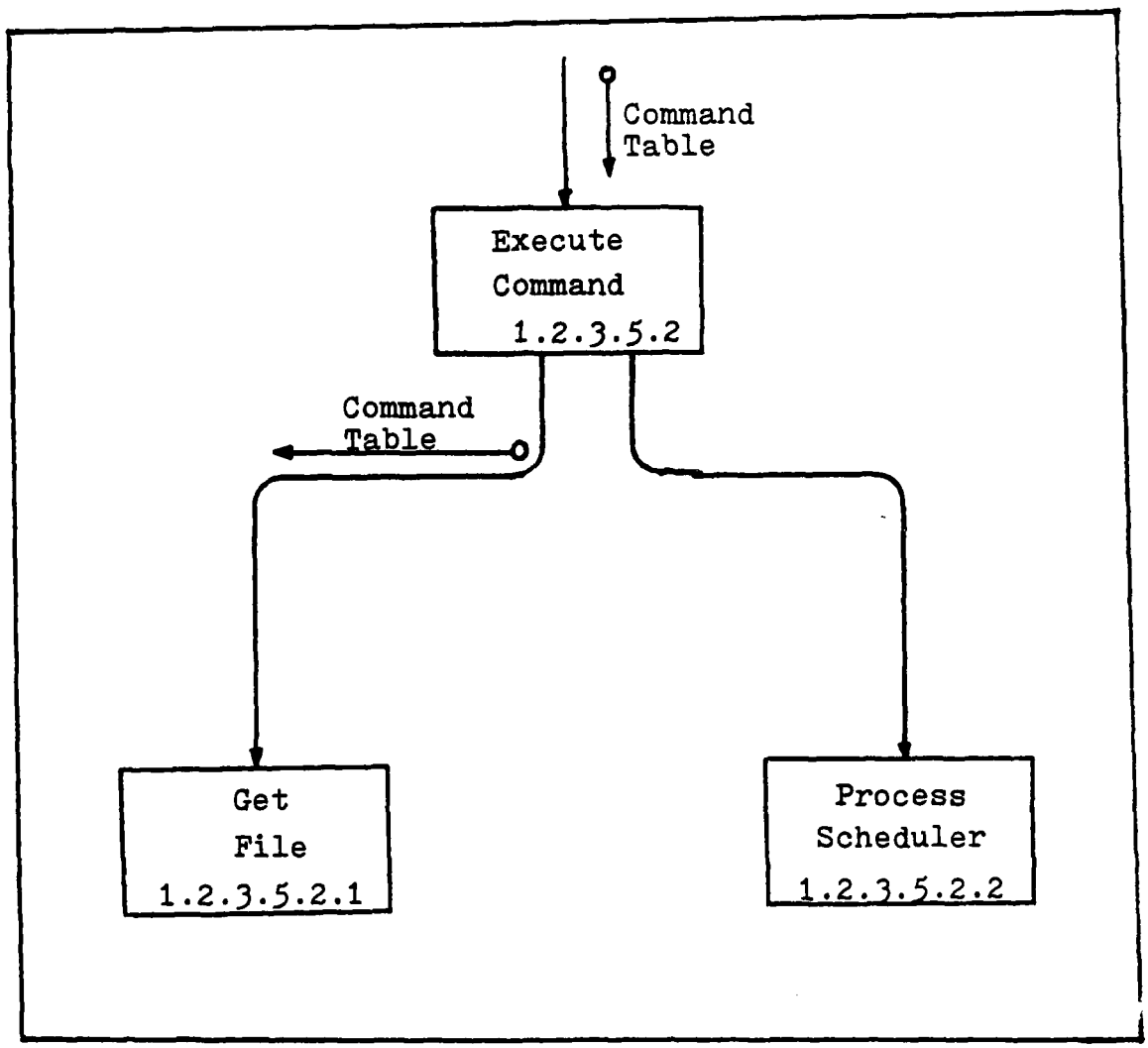


Figure IV-9 Execute Command

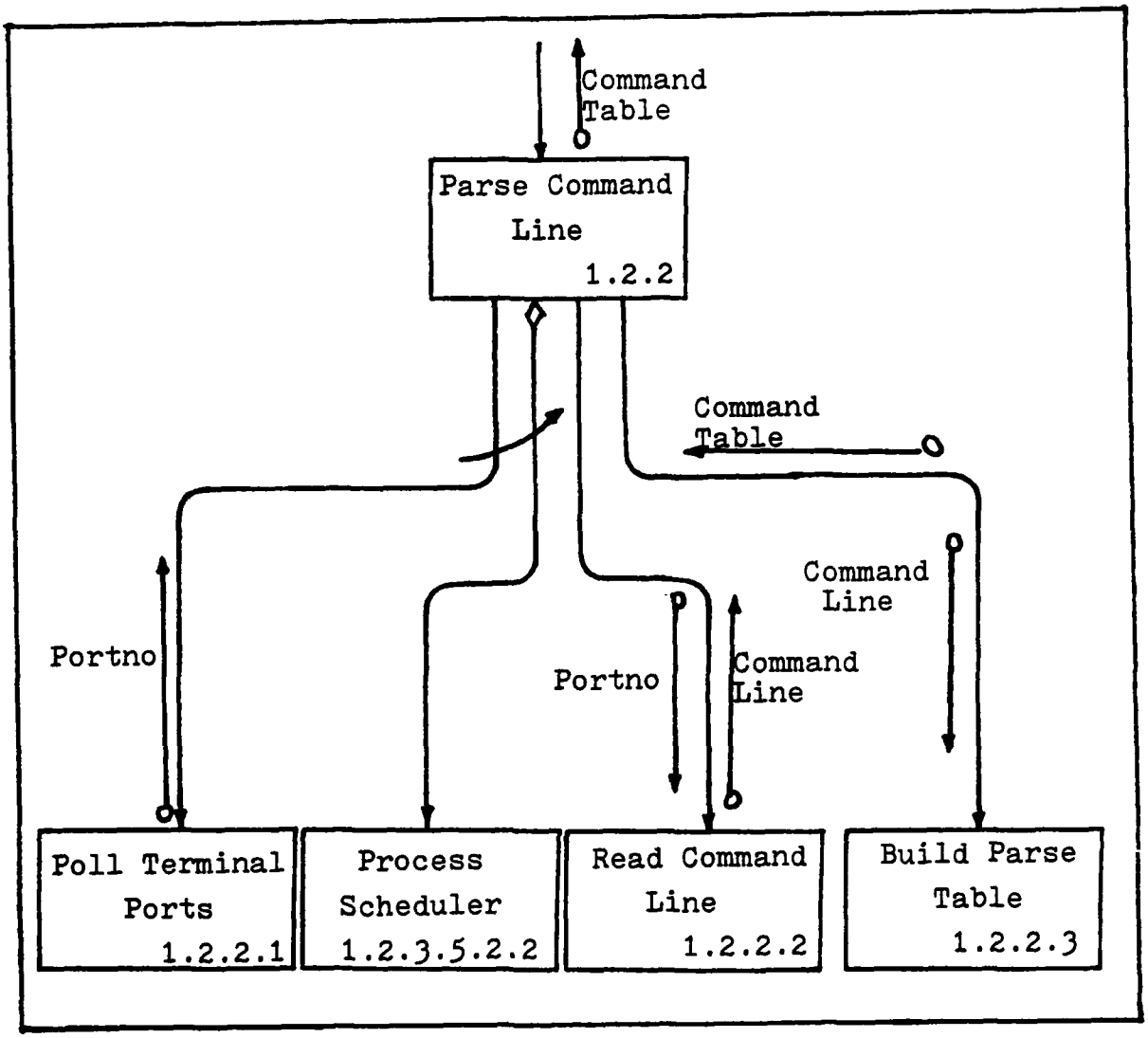


Figure IV-10 Parse Command Line

code. The pcbs are declared as an array. Since the users are only allowed one job running at a time, the array contains as many pcbs as there are user terminals. This makes allocating the pcb for the process easy; just allocate the arrays port number element (that is, for port number 1, allocate pcb[1]).

After allocation of a pcb consists of initializing the information that is contained in the pcb. THe information that is initialized is the priority of the process, the I/O status, the current queue that it is residing in, the command type of the processes origin, and the port of origin. After the pcb is built, the user terminal's user block's jobrunning is set to true.

The subroutine was coded using the design given for the BUILD PCB module in Figure IV-2. The following is the pseudocode of the Build PCB subroutine.

*****

```
Procedure Build PCB
 Set port of origin to port number
 Set I/O status to on
 If SYS command origin
  Set priority to 0
  Set current queue to 0
  Set I/O status to off
```

```
 Set command type to job code
End if
Else
 Set priority to 1
 Set current queue to 1
 Set command type to job code
End else
 Insert pcb into a queue
End Procedure Build PCB

*****
```

After the pcb is allocated and initialized, it must be inserted into a queue. This is done by using the doubly linked list method (DDL) (Ref. 13: 140). The DDL has easy insertion and deletion techniques for the elements. An alternative to the DDL is a static array of tables (array of pcb storage areas). The array would take more storage than a DDL, since it would have a set number of tables, empty or full, while the DDL would only have the number of tables residing in the queue. The transferring of the tables requires the duplication of the information in the array, while the DDL would only transfer the pointers of the tables to another DDL. The queues headers have a pointer to the first element and the last element and a counter to indicate how many pcbs are

in the queue. The new pcb is added onto the end of the queue by assigning the pointers in the correct order. After the pcb is inserted, the queue count is incremented.

The following is the pseudocode for the Insert PCB subroutine.

*****

```
Procedure Insert PCB
 If not in I/O wait state
  then
   If priority is 0
    then
     Insert pcb into System queue
     Increment the System queue counter
   End if
   Else
    If priority is 1
     then
      Insert pcb into Ready1 queue
      Increment the Ready1 queue counter
    End if
  End else
 End if
 Else
  Insert pcb into I/O Wait queue
```

```
  Increment the I/O Wait queue counter
 End else
End Procedure Insert PCB
```

*****

Process Scheduler

The process scheduler's implementation follows the design with no problems (see pages IV-8 to IV-15). The I/O Wait queue is processed determine to determine if there are any processes that need to be inserted into one of the Ready queues or the System queue. A pcb is retrieved from one of the queues, if there is a ready process, and the process is executed. The Run Process turns out to be implementable using the switch-case (Ref. 14: 74). The subroutine was coded using the design given for the PROCESS SCHEDULER module in Figure IV-3. The following is the pseudocode for the Process Scheduler subroutine.

*****

```
Procedure Process Scheduler
 Process the I/O Wait queue
 Check the queues
```

```
 Return if the queues are empty
 Case (queue with a ready pcb)
  Case System queue:
   Get first pcb
   Run System command
  End System queue case
  Case Ready1 queue:
   Get first pcb
   Case command type
    Case 0:
      Run Program
    End Case 0
    Case 1, 4:
      Send File
    End Case 1,4
    Case 5:
      Write
    End Case 5
    Default: command type error
   End Case command type
   Deallocate Space
  End Case Ready1 queue
 End Case (queue with a ready pcb)
End Procedure Process Scheduler

*****
```

In processing the I/O Wait queue, the whole queue is searched to find the processes that are finished with their I/O wait. The pcb is then inserted into the System queue or one of the Ready queues. The subroutine was coded using the design given for the PROCESS I/O WAIT QUEUE module in Figure IV-4. The following is the pseudocode of the Process I/O Wait Queue subroutine.

*****

```
Procedure Process I/O Wait Queue
 If I/O Wait queue counter is greater than 0
  then
   While not end of queue
    Set pointer to next pcb
    If I/O status is 0
     then
      Delete from the queue
      Insert pcb into ready queue
      Decrement the I/O Wait queue counter
    End if
   End while
 End if
End Procedure Process I/O Wait Queue
```

*****

The System queue and the Ready queues are checked to see if there are any ready processes. This is done by calling Check Ready Queues. This subroutine returns a value to the Process Scheduler to indicate either of three of the following:

1. The queues are empty
2. The System queue has a ready process
3. The System queue is empty and the Ready queue has a ready process.

The following is the pseudocode for the Check Ready Queues subroutine.

*****

```
Procedure Check Ready Queues
 If System queue counter is 0
  then
   If Ready queue counter is 0
    then
     Return the queues are empty
   End if
   Else
    Return System queue empty/Ready queue not
```

```
   End else
 End if
 Else
  Return System queue has ready process
 End else
End Procedure Check Ready Queues
```

*****

To run a system command process or a user command, the proper subroutines would have to be called from the Process Scheduler. This would only be the case if the process for a system command is sent through the Process Scheduler. For the LIST, PRINT, and DIR commands, the Send File subroutine, which is the I/O management routines that would send the information, is called. For the DEL command, the Write subroutine, which is the disk management routine that would send the new directory to the disk, is called.

If the process is an executable program that is to be run (a RUN command process), the subroutine Program Run is called. Running of an executable program requires the use of an assembly language procedure. This procedure was not written, but the pseudocode for the procedure follows:

*****

```
Procedure AMOSKERNEL
 Enable hardware interrupts
 Execute program by jumping to the beginning
 Process Interrupts
 Check the error flags
 Disable Interrupt
 Send error setting
End Procedure AMOSKERNEL
```

*****

After the process is completed or aborted, the Deallocate Space subroutine is called to free the memory space that was used by the process.

Summary

In this chapter the AMOS Scheduler was presented. The design was broken into the job scheduler and the process scheduler, with the traffic controller designed into the process controller. The scheduling techniques, preemptive and nonpreemptive, were discussed with the nonpreemptive scheduling technique chosen for the initial implementation of the scheduler.

implementation of the scheduler.

The design was broken into three steps:

1. Design of the Job Scheduler
2. Design of the Process Scheduler
3. Overall Design of the AMOS Scheduler

The implementation of the AMOS Scheduler was then presented in two parts: 1) Implementation of the Job Scheduler and 2) Implementation of the Process Scheduler. The implementation into the operating system was discussed with each part. The C source code for the AMOS Scheduler can be seen in Appendix E.

The code was tested using the static analysis method to ensure that the design was followed and to find any blatant syntax or logical errors. The test plan is presented in Chapters Two and Six, while the static analysis is presented in Chapter Five.

V. Implementation of the Operating System

Introduction

The purpose of this chapter is to present the pseudocode of the algorithms that were chosen for implementation in the detailed design. The coding of the operating system, and some of the problems that occurred, are also discussed.

"Coding is the implementation of the refined design, with the idiosyncracies of the programming language, operating system environment, and external (human and hardware) interfaces taken into account" (Ref. 11: 12). Since the structure charts were used in the detailed analysis, the coding was nearly a one-to-one transfer from structure chart module to coded subroutines. This provided the modularity that was striven for in the design. The updating of any of the subroutines, either making a minor change in the original subroutine or replacing the entire subroutine, can be done without changing any other routines, as long as the interfaces between subroutines do not change.

Main

The Main subroutine is the first procedure that is executed. This subroutine was coded using the design given for the EXECUTE AMOS module in Figure III-2 on page III-5. The operating system is centralized around this subroutine. The following is the pseudocode for the Main subroutine:

*****

```
Procedure Main
 Initialize Data Base
 Loop
  Parse the command line
  Determine valid command
 Forever
End Procedure Main
```

*****

Initialize Data Base

The Data Base is information used by the operating system and is defined in the source code. Since the passing of variable parameters is complex, the C-defined

structures for the Data Base are global to the operating system. The variables are made global by defining them before the main module (Ref. 12).

The implementation of initializing the Data Base is done by using a simple operator, the assignment or '=.' This subroutine was coded using the design given in Figure III-3, on page III-7. The information that will remain constant during the execution of the operating system is implemented using the C language '#define' (Ref. 12: 86). The information that might be changed during the execution of the operating system is initialized in a subroutine, called INITIALIZE-DATA-BASE (see Appendix E). The initial values can only be changed by software enhancement. This implementation is only recommended for the initial testing. Any updated version of INITIALIZE DATA BASE module should follow the design presented in figure III-3, on page III-7.

The suggested implementation for initializing the Data Base is reading the information from a file on the operating system's disk (e.g. the VMS O/S for the VAX 11/780 located in the Digital Engineering Lab (Ref. DEL)). This allows for an easy updating of the Data Base. For example, when adding another terminal to the computer system, it is not necessary to change the source code and recompile it. The changes can be done by a System command which can be written when this implementation is added.

The System command would change the information in the following parameters:

1. noports, the number of on-line terminal ports.
2. MAXJOBS, the maximum number allowed on the system.
3. portdata, the data table for the terminal ports.

The changes in the Data Base would be saved in the Data Base file either when the System command to change the Data Base is executed or when the operating system is shut down. The shutting down of the operating system would be performed by another System command that would also be written when this implementation is added. This System command is not needed for the initial implementation because the saving of the Data Base is not essential. The reason for this is the Data Base initialization is coded. It is recommended that the Data Base be saved at the time it is changed and at the time a system shut down is performed. This will ensure that the information of the Data Base is saved if a power failure occurs after a change.

Parse Command Line

This subroutine waits for a user to attempt to communicate with the operating system, gets the user's command line, and parses the line. The command line is the character string that the user inputs from the terminal and is terminated by a carriage return. This subroutine was coded using the design given for the PARSE COMMAND LINE module in Figure III-4, on page III-9. The following is the pseudocode for the Parse Command Line subroutine:

*****

```
Procedure Parse Command Line
 If Poll is true
  then
   Get the command line
   Build the command table (parse the command line)
 End if
End Procedure Parse Command Line
```

*****

Poll

The Poll subroutine returns a boolean value (true or false). It polls the terminal ports to determine if there is any input. The polling algorithm used is circular, that is, it starts from the beginning of the ports (port 0), goes through to the ending of the ports (port n-1, where n is the number of terminals on line), and goes back to the beginning of the ports. It checks those ports that do not have a submitted job. If a response is found, the polling routine is stopped and the true is returned to the Parse Command Line subroutine. If it goes through one pass of the ports and does not receive a response, then it calls the Process Scheduler. The Process Scheduler takes care of the processes that are submitted. This ensures that any process that is in any of the process queues has the chance to run. After returning from the Process Scheduler, the polling is resumed. The following is the pseudocode for the Poll subroutine:

```
*****

Procedure Poll
 Set i to 0
 While no response
  While not one pass and no response
   If no process submitted from port[i]
    then
     Check port[i] for response
     If response
      then
       Return true
     End if
   End if
   Set i to next port number
  End while
  If no response
   then
    Process Scheduler
 End while
End Procedure Poll

*****
```

Determine Valid Command

The subroutine Determine Valid Command determines what type the command is (i.e. system, user, help, and control). If that particular command is valid, then it calls the necessary routines to have the command executed. This subroutine was coded using the design given for the DETERMINE VALID COMMAND module in Figure III-5, on page III-10. The following is the pseudocode for Determine Valid Command:

*****

```
Procedure Determine Valid Command
 If user already logged in
  then
   If Log out command
    then
     Log Out User
   End If
   If Help command
    then
     Help User
   End if
   If System command
    then
```

```
        System Change
      End if
      If User command
       then
        Execute User Command
      End if
      If invalid command
       then
        Send invalid command message
      End if
   End if
End Procedure Determine Valid Command
```

*****

Log In User

The procedure Log In User accomplishes one or two functions. If a user inputs a command line, then Determine Valid Command wants to know if the user is already logged on. If the user is logged on, then control returns to Determine Valid Command to execute the user's command, else the user is attempted to be logged on. The global table userblock has a logged on flag that is set to true or false. The flag that corresponds to the terminal's port number is checked to determine if the user

is logged on, yet. This subroutine was coded using the design given for the LOG-IN USER module in Figure III-7, on page III-15. The following is the pseudocode for the Log In User subroutine:

*****

```
Procedure Log In User
 If user not logged on
  then
   Prompt user for username
   Read in username
   Prompt user for password
   Read in password
   If legal username and password
    then
     Copy username into userblock
     Set loggedon flag
     Increment number of users on system
     Send login completed message to user
   End if
 End if
End Procedure Log In User
```

*****

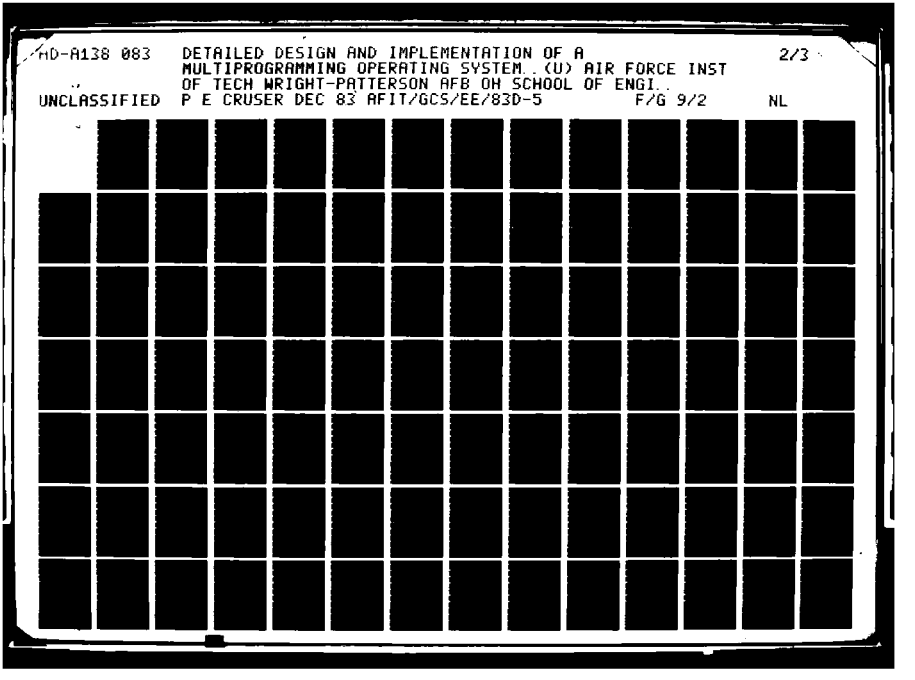
Log Out User

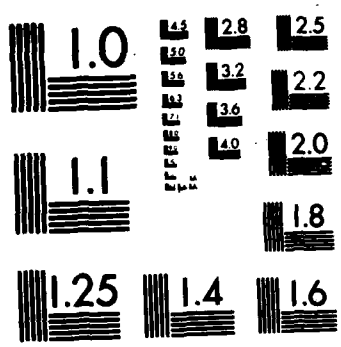
The Log Out User subroutine is a simple routine, but accomplishes an important cleanup function. It clears the userblock of the username and sets the logged on flag to false. It also sets the jobrunning flag to false. The jobrunning flag should already be false, since the user cannot communicate with the operating system unless no jobs were running. This subroutine was coded using the desgn given for the LOG-OUT USER module in Figure III-8, on page III-17. The following is the pseudocode for the Log Out User subroutine:

*****

```
Procedure Log Out User
 Send logged out message to user
 Clear username space in userblock table
 Set loggedon and jobrunning to false
 Decrement number of users on system
End Procedure Log Out User
```

*****


AD-A138 083
DETAILED DESIGN AND IMPLEMENTATION OF A MULTIPROGRAMMING OPERATING SYSTEM..(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI..
2/3
UNCLASSIFIED
P E CRUSER DEC 83 AFIT/GCS/EE/83D-5
F/G 9/2
NL



Help User

The Help User subroutine provides information to the requesting user. The two types of information that is provided are system information and command information (Ref. 4: 52). A system help request would give information to the user about the system. The following is an example of a system help request:

'HELP USERS'

This command line would result in the listing of all of the users that are logged into the system and what terminal number that each user is using. A command help request would give the format for the command and any information for the command.

The following is an example of a command help request:

'HELP DEL'

The response from the operating system would be as follows:

'Format: DEL FILENAME'

This subroutine was coded using the design given for the HELP USER module for in Figure III-9, on page III-19. The following is the pseudocode for the Help User subroutine:

*****

```
Procedure Help User
 If system information is requested
  then
   Determine what information is requested
   Get the information and send to the user
 End if
 Else
  If command information is requested
   then
    Determine which command's information is needed
    Send the format and other information
  End if
  Else
   Send no help available message
  End else
 End else
End Procedure Help User
```

*****

System Change

The System Change subroutine determines that the user is the 'Superuser', then it determines what changes in the Data Base are requested, gives the user any needed prompts for the new information, and performs the necessary changes. This subroutine was coded using the design given for the SYSTEM CHANGE module in Figure III-6, on page III-13. Because the initial coding of the operating system does not have a dynamic Data Base (that is, the initial values cannot be changed from loading to loading on the computer system), there are no system changes that can be permanently performed. When the dynamic Data Base, using a Data Base file, is implemented, system changes can be performed with the changes saved.

In the coding and design of this operating system, a process is sent through the Process Scheduler to be executed. But the system command is an exception, it is executed without having to go through the Process Scheduler to run. The option to execute it with the other processes is in the source code, because of the following:

1. A System Ready Queue is available.
2. A Process Control Block (pcb) is made for the system command.

3. The pcb is entered into the System Ready Queue.

4. A subroutine is written that would execute the system command when it is given the processing time.

The following is the pseudocode for the System Change subroutine:

*****

```
Procedure System Change
 If the Superuser
  then
   Determine which change is to be performed
   Prompt the Superuser for the new information
   Get Superuser's response
   Change the Data Base
 End if
 Else
  Send not Superuser message to user
 End else
End Procedure System Change
```

*****

Execute User Command

This subroutine determines if the requested user command is valid. That is if the file being requested is located in secondary memory, and if the user is requesting their file. This subroutine was coded using the design given for the EXECUTE USER COMMAND module in Figure III-10, on page III-21. For the RUN command it is also necessary to determine if the requested file is an executable file. If the command is found to be valid then it calls the necessary routine for execution. The following is the pseudocode for the Execute User Command subroutine:

```
*****

Procedure Execute User Command
 If the command is valid (Validate User Command)
  then
   Execute the valid command (Execute Command)
  End if
  Else
   Send error message to the user
  End else
End Procedure Execute User Command
*****
```

Validate User Command

This subroutine determines if the user has input valid parameters. If all the parameters are valid, control is returned to the calling subroutine. If any parameter is invalid, an error subroutine is called to handle the particular error.

This subroutine was coded using the design given for the VALIDATE USER COMMAND module in the figure on page B-11. The following is the pseudocode for the Validate User Command subroutine:

*****

```
Procedure Validate User Command
 If RUN command
  then
   Check filename, username, and if executable file
 End if
 If LIST, PRINT, or DEL command
  then
   Check filename, check username
 End if
 If DIR command
  then
```

```
   This command is always valid
 End if
End Procedure Validate User Command
```

*****

Execute Command

This subroutine calls the necessary subroutine to move a job from secondary memory into main memory. The steps to perform this task are:

1. Locating the file.
2. Getting the file.
3. Placing the file into main memory.
4. Calling the appropriate subroutine.

After these steps are finished the Process Scheduler is called. The Process Scheduler will then perform the appropriate steps to execute the specified command. This subroutine was coded using the design given for the EXECUTE COMMAND module in the figure on page B-19.

Build Message

The Build Message subroutine builds the required message that is transmitted to the user. These messages fall into the following three categories:

1. Prompts.
2. Command formats.
3. System messages.

A prompt is a message sent to the user that is requesting some additional information. This information can consist of username, password, Data Base changes, and others. A command format is a message that informs the user of the required format for a command. This message is used by the Help User subroutine. A system message is a message sent to inform the user that a system change has been completed.

The calling subroutine sends a single coded parameter used in the seletion of the message that is to be sent to the user. The message is not actually 'built,', but it is defined in the beginning of the subroutine. The message is then sent to the Transmit Message subroutine, which needs the terminal's port number that the message is to be sent.

This subroutine was coded using the design given for

the BUILD MESSAGE module in the figure on page B-24. The following is the pseudocode for the Build Message subroutine:

*****

```
Procedure Build Message
 Define Messages
 Case message code
  Code = 0: Send no help message
  Code = 1: Send run command format
  Code = 2: Send list command format
  Code = 3: Send print command format
  Code = 4: Send delete command format
  Code = 5: Send directory command format
  Code = 6: Send username prompt
  Code = 7: Send password prompt
  Code = 8: Send logged out message
  Code = 9: Send login complete message
  Code =10: Send job done message
  ( additional messages can be added )
  Default: Send no message error (for testing purposes)
 End case
End Procedure Build Message
```

*****

Error Handling

Errors are handled through a subroutine called 'Error', and then control is returned to each calling subroutine indicating an error was received. Having control returning to each calling subroutine indicating that an error was received, allows for the errors to be handled efficiently. The Error subroutine performs the same type of function as the Build Message subroutine does, except the messages that are sent are error messages. This means that the error messages are defined in the beginning of the subroutine

This subroutine was coded using the design given for the BUILD ERROR MESSAGE module in the figure on page B-18. The following is the pseudocode for the Error subroutine:

*****

```
Procedure Error
 Define error messages
 Case error message code
  Code = 1: Send syntax error received
  Code = 2: Send invalid filename
  Code = 3: Send improper user retreiving file
  Code = 4: Send illegal user trying to log in
```

```
    Code = 5: Send unauthorized user attempting to do
              system change
    Code = 6: Send unreconizeable was received
    Code = 7: Send not enough space to execute job at this
              time
    Code = 8: Send program too large for executi
    Code = 9: Send process was aborted before co   .ion
    Code = 10: Send Non-Executable file, unable to run
    Code = 11: Send Executable file, unable to print
(additional error message can be added when necessary)
    Default : Send no error message error (for testing
              purposes)
  End case
End Procedure Error

*****
```

Static Analysis

Testing the source code requires using a software testing technique. The initial testing on the AMOS source code was static analysis. Static analysis is "a collection of analysis and testing methods that do not require the execution of the subject program" (Ref. 24: 5-1). The capablities that static analysis can accomplish are: (Ref. 24: 5-1)

1. Detect and locate certain types of program errors.
2. Identify program anomalies, characteristics that produce errors.
3. Identify constructions that do not conform to the standard syntax.
4. Determine whether the variables are used in accordance with the intentions of the programmer.
5. Help to generate test data for dynamic testing.

The types of program errors that are looked for are infinite loops, module interface conflicts, recursive procedure calls, and uninitialized variables (Ref. 24: 2-2). Because the design was evaluated closely, the only program errors that were found were uninitialized variables. This was the result of a programmer forgetting to initialize counters used in conditional statements and loops.

One program anomaly was found, using static analysis, and corrected in the design, as well as the code. This anomaly was the deleting of two seperate files on the same sector that are on two seperate processes that are in ready states. The original design and code would delete the first file, but when the next file is deleted, the first file is restored and only the latter file is

deleted. This was resolved by making the DEL command dedicated. That is, the command is executed without interruptions. This ensures that a second DEL command does not negate the first DEL command.

The syntax was tested using the C language compiler. The compiler used is on the VAX 11/780 VMS O/S. This compiler only compiled the program and checked the syntax for the VAX C language, not for the standard syntax for the C language. When this operating system is transferred to another computer system, the source code will have to be recompiled for that computer's version of the C language.

A variable detected that was not used as it was intended was 'end.' 'End' was defined in the global variables as a right bracket. In the subroutine Get-file, the variable 'end' was used as a flag that indicated an end of the file marker was found.

The test data that can be used for the dynamic testing of AMOS could be the scenario inputs that were used in the static analysis. These scenario inputs are:

1. RUN 'any parameter'
2. LIST 'any parameter'
3. PRINT 'any parameter'
4. DEL 'any parameter'
5. DIR

6. SYS 'any parameter'

7. HELP 'any parameter'

8. BYE

9. 'any parameter'

'Any parameter' is input that is valid or invalid. This would test the valid cases and the invalid cases of each command. A carriage return would be considered as a parameter.

Summary

This chapter presented the alogrithms and pseudocode for the modules implemented. AMOS is mainly constructed with these modules. The pseudocode was translated into a source code language, C. The source code was developed on the VMS Operating System's EDT editor, which is on the VAX 11/780 located in the Digital Engineering Lab.

VI. Conclusions and Recommendations

Conclusions

This thesis effort was concerned with the detailed design and implementation of a multiprogramming operating system for sixteen-bit microprocessors. The detailed design consisted of reviewing the defined system requirements (Ref. 4 and 5) and following the top-level design specifications, in the form of data flow diagrams, to construct the detailed design. The implementation consisted of transferring the detailed design into a structured language, that is the C language.

The detailed design looked at the single user environment to determine the processes that would be necessary for the operating system. This was the majority of the operating system design (Chapter 3). The design was constructed for a single user and was modified to handle a multi-user environment. These modifications consisted of inserting a scheduler and a memory manager.

The implementation of AMOS followed the design, except for one detail. Global variables were used instead of passing parameters between modules, because passing structure values on the stack is impossible in C (Ref. 1: 66). The only variables passed between modules are flags

and a few others, such as track locations, sector locations, and command types. The problem of passing structure values on the stack was also encountered in a previous effort (Ref. 1: 66).

In this effort, the detailed design and implementation of the AMOS scheduler was done. The detailed design consisted of reviewing the defined scheduler requirements (Ref. 5: 72) and following the top-level design specifications, in the form of data flow diagrams (Ref. 5: 121-126), to construct the detailed design. The implementation used the same method that the rest of the operating system used, i.e. transferring the detailed design into a structured language.

The objective that was not met was the actual execution of the scheduler and the operating system. The scheduler, as was the operating system, was tested using the static analysis technique and was found to be logically correct. It has not been tested using the test plan given at the end of Chapter 2. Logically, the scheduler will provide a multiprogramming environment for the operating system.

Recommendations

This effort does not complete the software development cycle for AMOS. The testing phase and the coding of assembly language subroutines are not completed. Therefore, the source code has to be transferred from the VMS O/S file system to a compatible 8 inch floppy disk for the Am Z8000 system. Also, the Z8000 system must be operational with a disk system, before the transfer can take place and installation of AMOS can occur. Follow-on thesis efforts are recommended to complete the operating system's software development cycle.

Testing

The static analysis of the source code has already been completed leaving dynamic testing to be performed. This dynamic testing should include module, integration, system, and acceptance testing (Ref. 25: 232). The completions of these tests should be extensive to insure a working product. It is recommended that these tests should be done before transfer to the microcomputer system, because the availability of software tools, for example, a compiler and an editor.

Module testing is the validation of a single module, usually isolated from all other modules (Ref. 25: 232).

This is done by using stubs in place of any modules that is called by the module being tested and, also, by using a driver to execute the module.

After completion of module testing, integration testing should be performed. Integration testing is a validation of the interfaces between modules, components, and subsystems (Ref. 25: 232). This testing should be done in a top-down fashion in order to prevent errors from propagating down to lower-level modules. If the testing is done in another fashion, an error that is found in a higher-level module interface will most likely propagate to lower-level modules that have already been integrated.

System testing is the validation of the system to its initial objectives: it is a validation process when done in a simulated environment or done in a live enviroment (Ref. 25: 232).

Acceptance testing is the validation of the system to the user requirements (Ref. 25: 232), which are defined in Chapter 2.

Assembly Coded Routines

The assembly coded routines necessary for an operational system are the device drivers and a kernel for the scheduler. The device drivers might be obtained from existing software, such as an existing operating system

for the microcomputer. The kernel will have to be written, because it will have to meet the specifications and design of the scheduler.

Source Code Transfer

Currently the source code is located on the VMS O/S's disk storage and will have to be transferred to an 8 inch floppy disk. This can be done by performing the following steps:

1. Transfer the source code onto a magnetic tape from the VMS disk storage.
2. Transfer the source code from the magnetic tape to UNIX disk storage.
3. Make any necessary syntax changes that makes the code compatible with the UNIX O/S.
4. Cross-compile the source code from C to Z8000 assembly code by using the cross-complier available for the UNIX O/S (Ref. 1: 67).
5. Transfer the Z8000 assembly code to an 8 inch floppy disk that has a compatible format for the Z8000 system.

Operational Z8000 System

For the system to be operational, the hardware components must be compatible and connected, and a developmental operating system with a Z8000 assembler must be obtained. The developmental operating system is needed to execute the Z8000 assembler, so that the AMOS assembly code can be converted to executable code.

Bibliography

1. Huneycutt, Douglas S. Design a Multiprocessing Operating System for Sixteen Bit Microprocessor, MS Thesis, Wright-Patterson AFB, Ohio: School of Engineering, Air Force Institute of Technology, December 1982.

2. Shaw, Alan C. The Logical Design of Operating Systems. New Jersey: Prentice-Hall, 1974.

3. Kaisler, Stephen H. The Design of Operating Systems for Small Computer Systems, New York: Wiley-Interscience, 1983.

4. Ross, Mitchell S. Design and Development of a Multi-programming Operating System for Sixteen Bit Micro-processors, MS Thesis, Wright-Patterson AFB, Ohio: School of Engineering, Air Force Institute of Technology, December 1981.

5. Yusko, Robert J. Development of an 8086 Multi-programming System, MS Thesis, Wright-Patterson AFB, Ohio: School of Engineering, Air Force Institute of Technology, December 1981.

6. Dzida, W. et al. "User Perceived Quality of Interactive Systems," IEEE Transactions on Software Engineering, Vol SE-4, No.4, 270-276 (July 1978).

7. Madnick, Stuart E. and John J. Donovan, Operating Systems, New York: McGraw-Hill, 1974.

8. Richie, D.M. "A Retrospective," Bell System Technical Journal, 57: 1947-1970 (July - August 1978).

9. Zelkowitz, Marvin V. "Perspectives on Software Engineering," Computing Surveys, Vol 10, No. 2 June 1978.

10. Fredman, Peter. Software System Principles - A Survey, Science Research Association, Inc., (1975).

11. Peters, Lawrence J. Software Design: Methods & Techniques, New York: Yourdon Press, 1981.

12. Coffman, E. G. Jr., et. al. Computer and Job-shop Scheduling Theory, New York: Wiley-Interscience, 1976.

13. Horowitz, Ellis and Sartaj Sahni. Fundamentals of Data Structures, Maryland: Computer Science Press, Inc., 1976.

14. Kernighan, Brian W. and Dennis M. Ritchie. The C Programming Language, New Jersey: Prentice-Hall, Inc., 1978.

15. Programming in VAX-11 C, Massachusetts: Digital Equipment Corporation, 1982.

16. Yourdon, Edward. Techniques of Program Structure and Design, New Jersey: Prentice-Hall, Inc., 1975.

17. Titus, Christopher A., et. al. 16-Bit Microprocessors, Indiana: Howard W. Sams & Co., Inc., 1981.

18. Zarrella, John, et. al. Microprocessor Operating Systems, California: Microcomputer Applications, 1981.

19. Hogan, Thom. Osborne CP/M User Guide, California: Osborne/McGraw-Hill, 1982.

20. Lions, J. UNIX Operating System Source Code Level Six, Bell Laboratories, 1977.

21. AM 96/4116 AmZ8000 16-Bit Monoboard Computer, California: Advanced Micro Devices, 1980.

22. "Software Installation Guide: System Management Operation," VAX/VMS, Vol. 10.

23. EE4.45 Lab Instuctions, Wright-Patterson AFB, Ohio: School of Engineering, Air Force Institute of Technology, October, 1983.

24. Heidler, et. al. Software Testing Measures, Griffiss Air Force Base, NY: ROME Air Development Center, AFSC, May, 1982.

25. Shooman, Martin L. Software Engineering: Design, Reliability, and Management, New York: McGraw-Hill, 1983.

Appendix A

Initial Hardware Configuration

The initial hardware configuration for the AFIT Multiprogramming Operating System (AMOS) is based on two factors:

1. The requirements that were defined for the microprocessor's computer sytem.
2. The availability of the microprocessor and the compatible hardware necessary to construct the computer system.

The microprocessor was selected earlier (Ref. 1) and is the AMD Z8002. The selection of the Z8002 was discussed on page II-14. The Digital Engineering Lab has an available AMD Z8002 microprocessor based computer system which consists of the following:

1. Heathkit terminal (1)
2. Heathkit H27 Floppy Disk Drives (Double)
3. Am 96/4116A Monoboard with Z8002 microprocessor
4. Am 95/6120 Intelligent Floppy Disk Controller
5. Am 96/1128 128K Dynamic RAM Board
6. Am 95/5132 RAM-EPROM-I/O Board

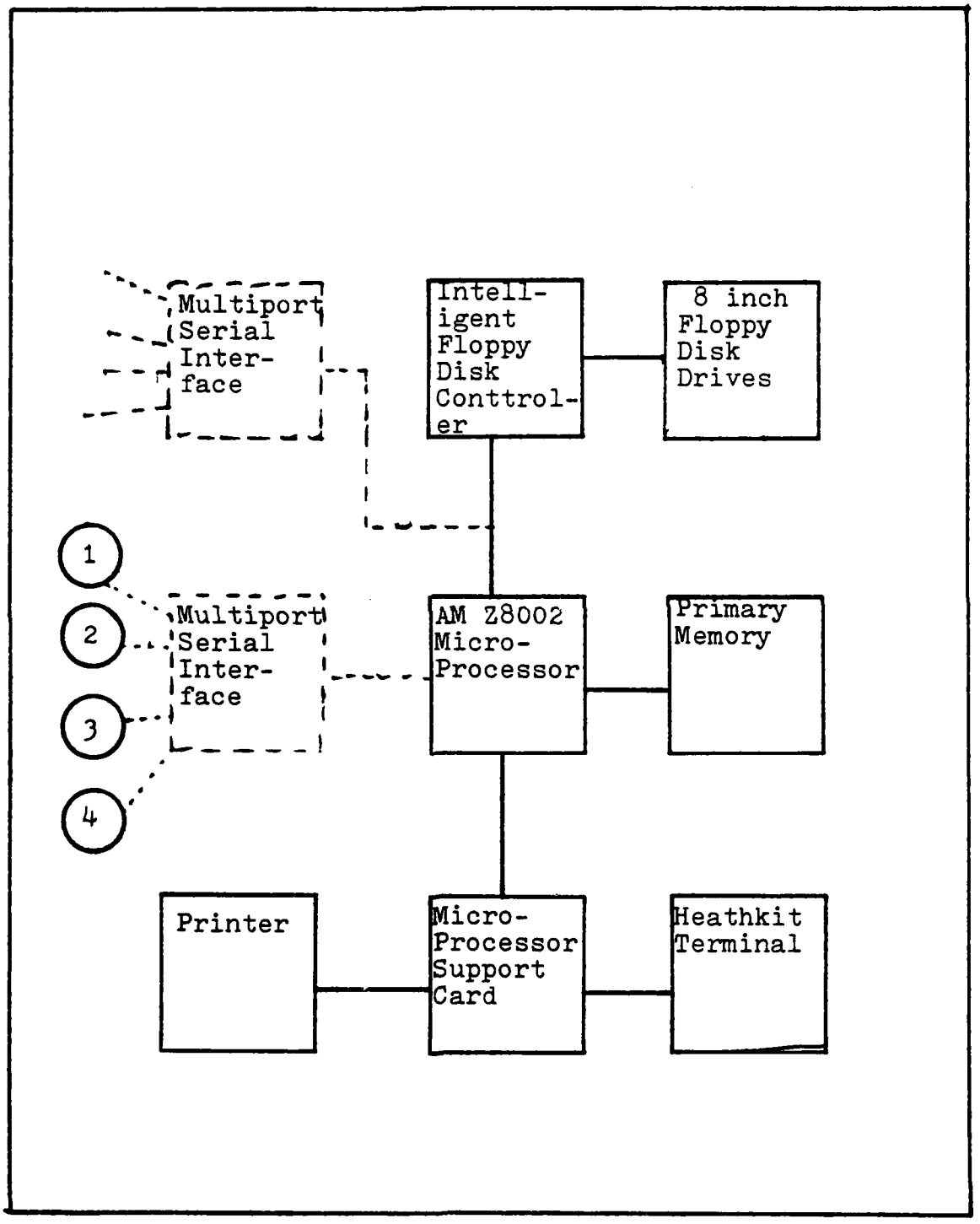
7. Amc 95/6011 Arithmetic Processing Unit Board
8. Am 95/6452 Card Cage

For the implementation of a multiuser environment, it is recommended that the standard multibus serial interface card should be incorporated with the above hardware. More terminals would also be required. Other units or peripherals can be added to the system when the need, or opportunity, arises.

The Am 96/4116 Monoboard contains two RS232 serial I/O ports, 24 parallel I/O lines, five programmable counter/timer at 4 MHz, and power-fail capability (Ref. 21:1). Further specifications can be found in Reference 21. The specifications for the other AMD products are in the following manuals:

1. Am 95/6120 Dual Density Floppy Disk Controller User's Manual
2. Am 96/1000 Series Dynamic Random-Access Memory Boards User's Manual
3. Am 95/5132 PROM/ROM/RAM and I/O Board User's Manual
4. Amc 95/6011 Arithmetic Processing Unit Board User's Manual
5. Am 95/6452 Card Cage User's Manual

The following is a diagram of the initial hardware configuration:

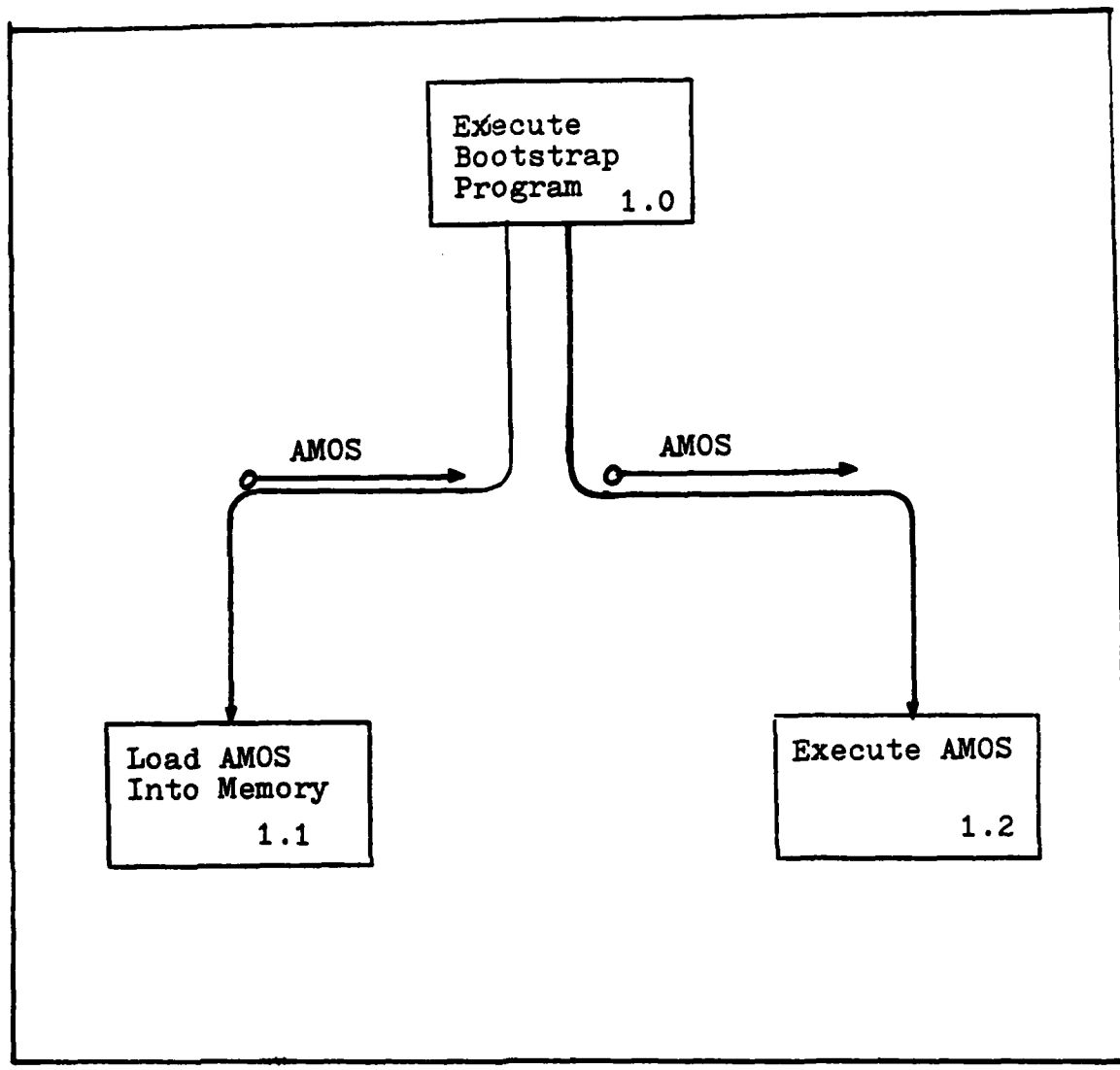


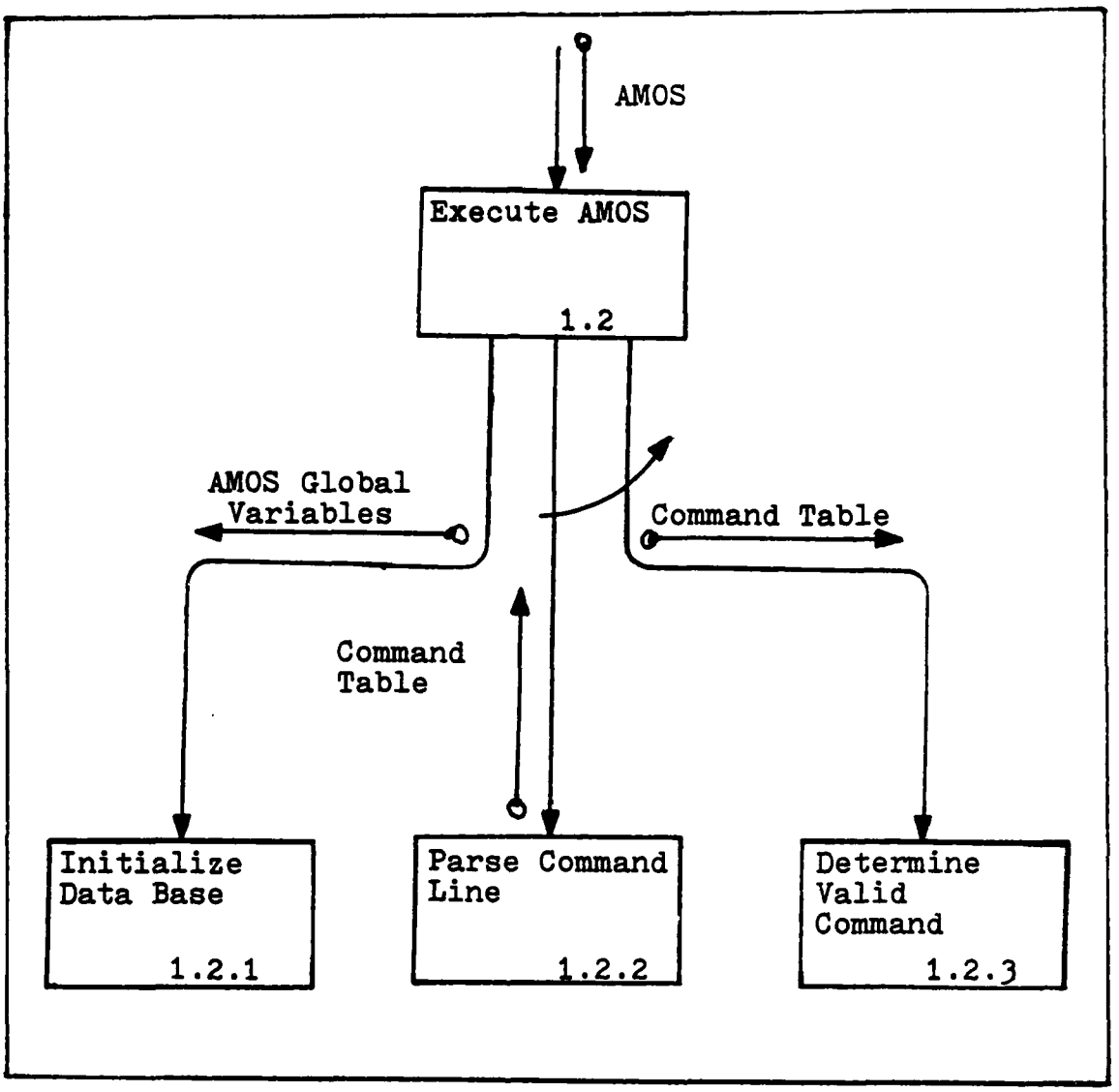
Appendix B

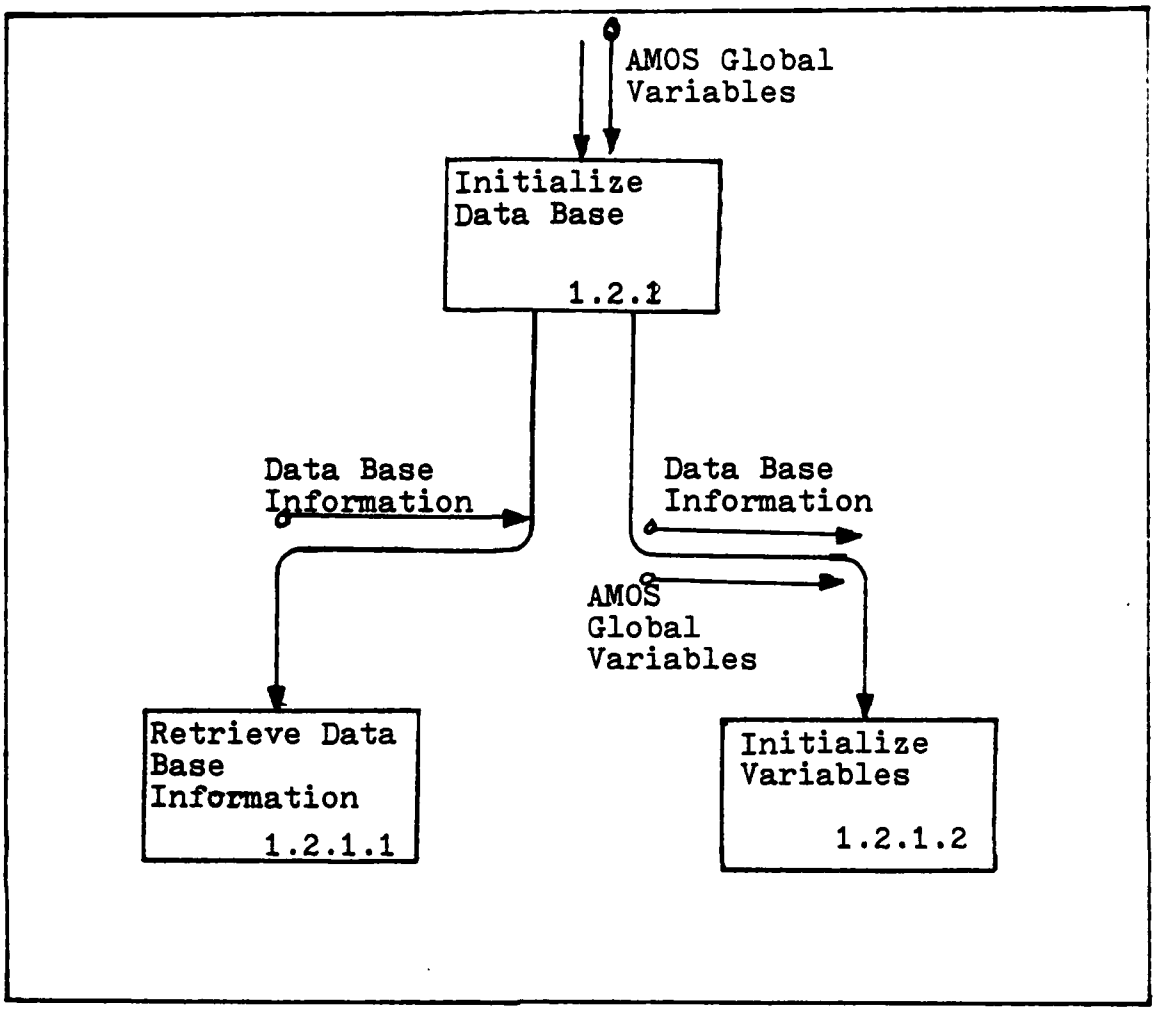
AMOS Structure Charts

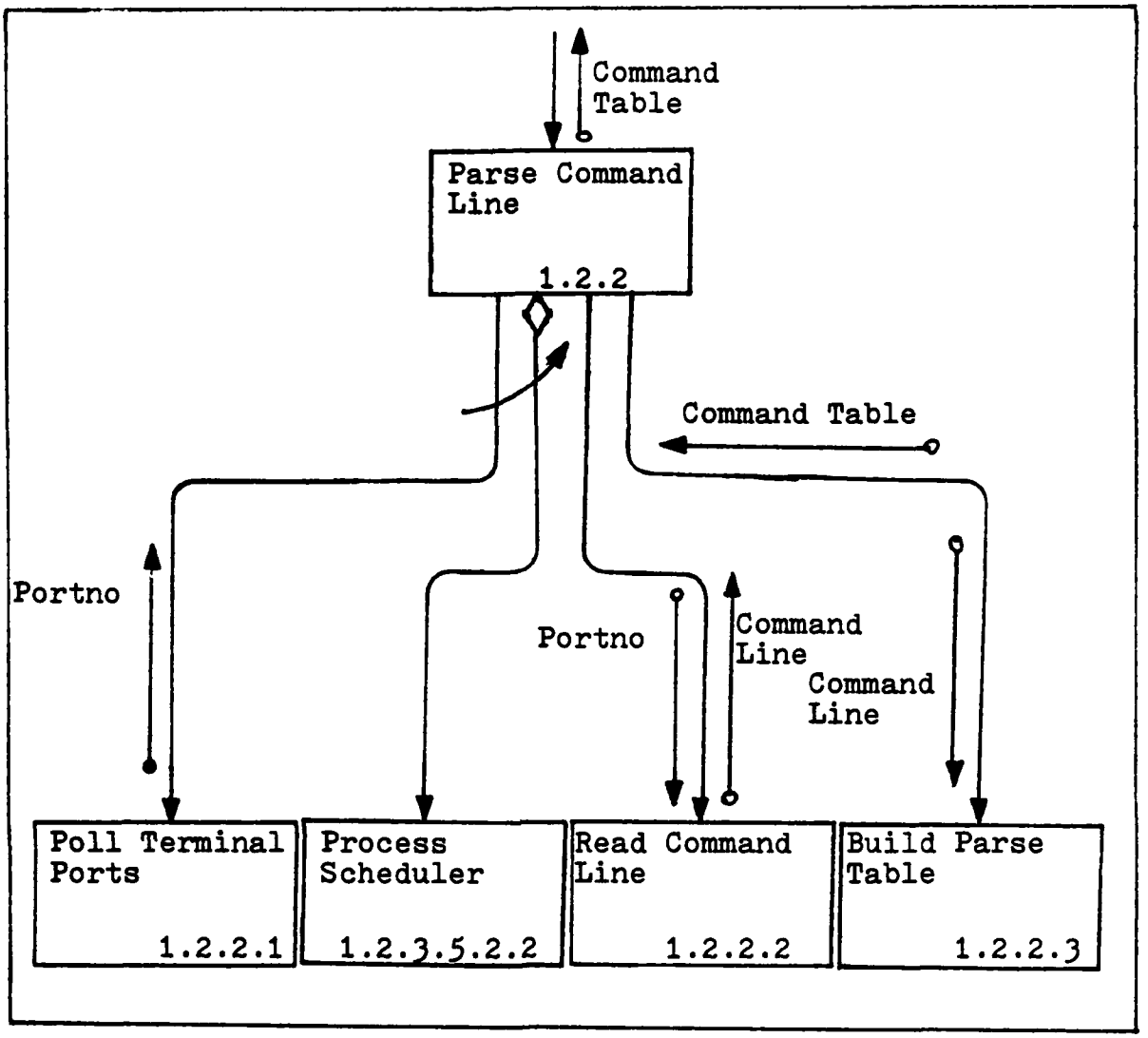
This appendix contains the structure charts for AMOS. The structure charts contain the modules designed in Chapter 3 and both the modules designed for the Memory Management and Process Management. The process descriptions and data flow entries are located in Appendices C and D, respectively.

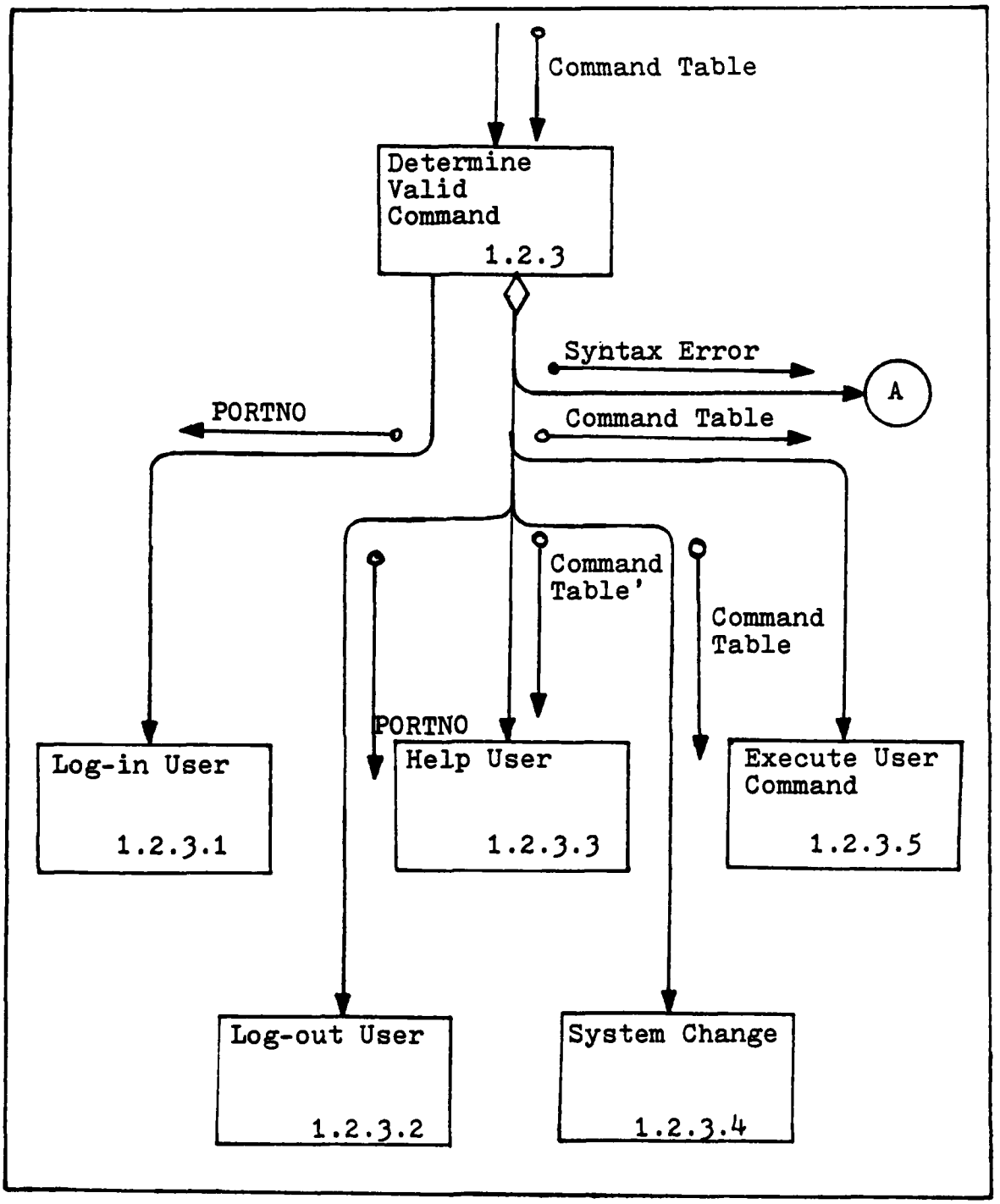
Index


Execute
Bootstrap
Program
1.0
AMOS
AMOS
Load AMOS
Into Memory
1.1
Execute AMOS
1.2

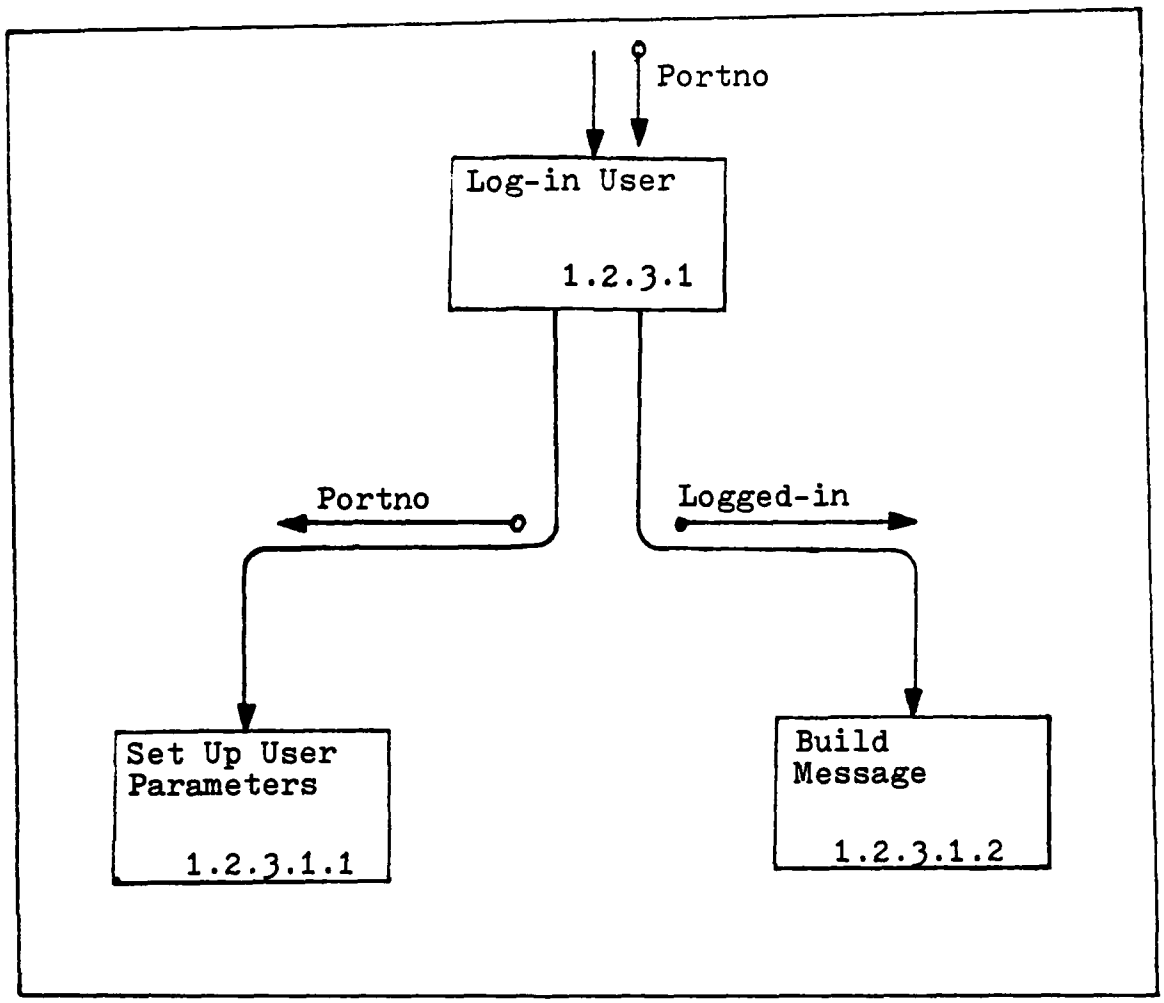

AMOS
Execute AMOS
1.2
AMOS Global Variables
Command Table
Command Table
Initialize Data Base
1.2.1
Parse Command Line
1.2.2
Determine Valid Command
1.2.3

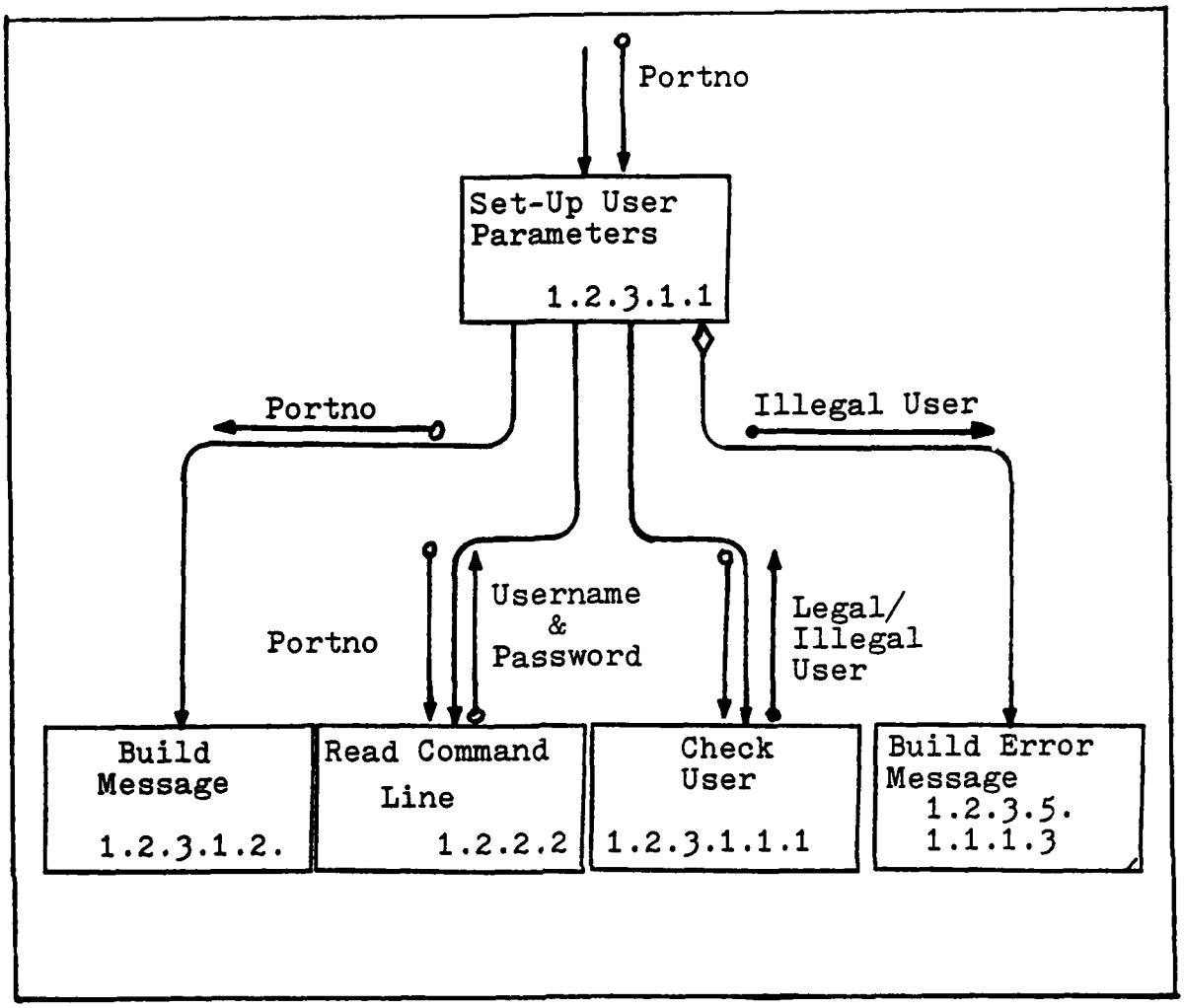

AMOS Global Variables
Initialize Data Base
1.2.1
Data Base Information
Data Base Information
AMOS Global Variables
Retrieve Data Base Information
1.2.1.1
Initialize Variables
1.2.1.2

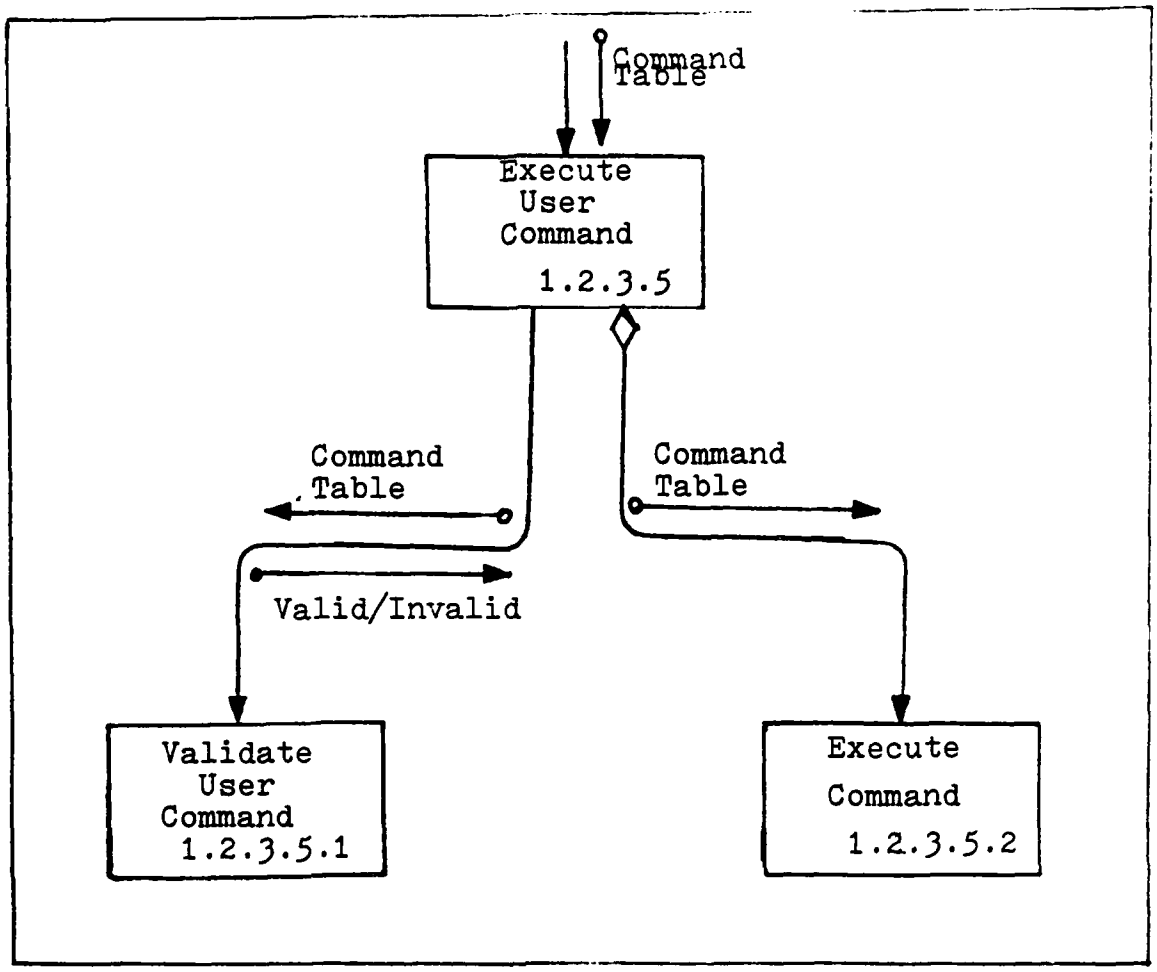

Command
Table
Parse Command
Line
1.2.2
Command Table
Portno
Portno
Command
Line
Command
Line
Poll Terminal
Ports
1.2.2.1
Process
Scheduler
1.2.3.5.2.2
Read Command
Line
1.2.2.2
Build Parse
Table
1.2.2.3

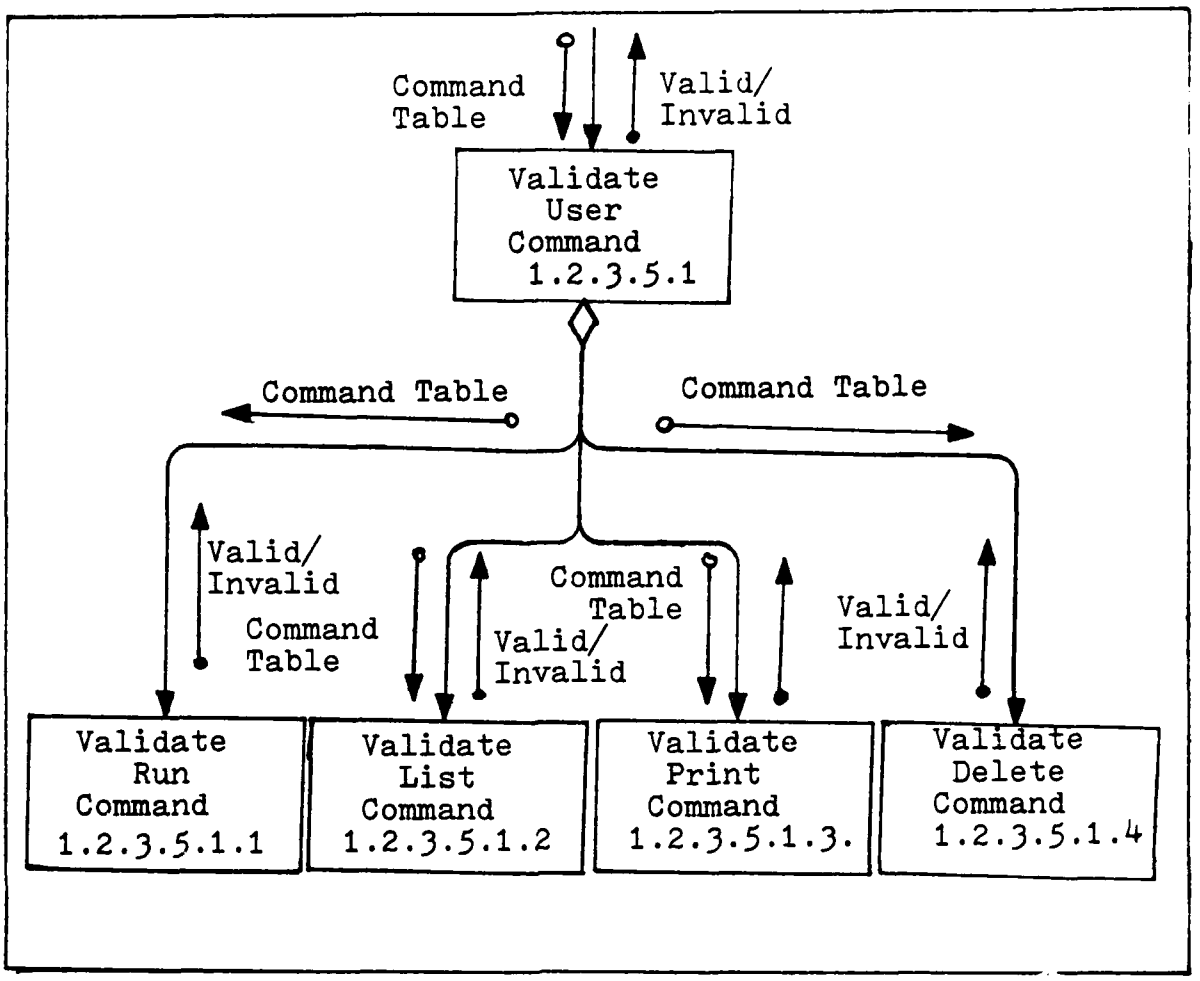


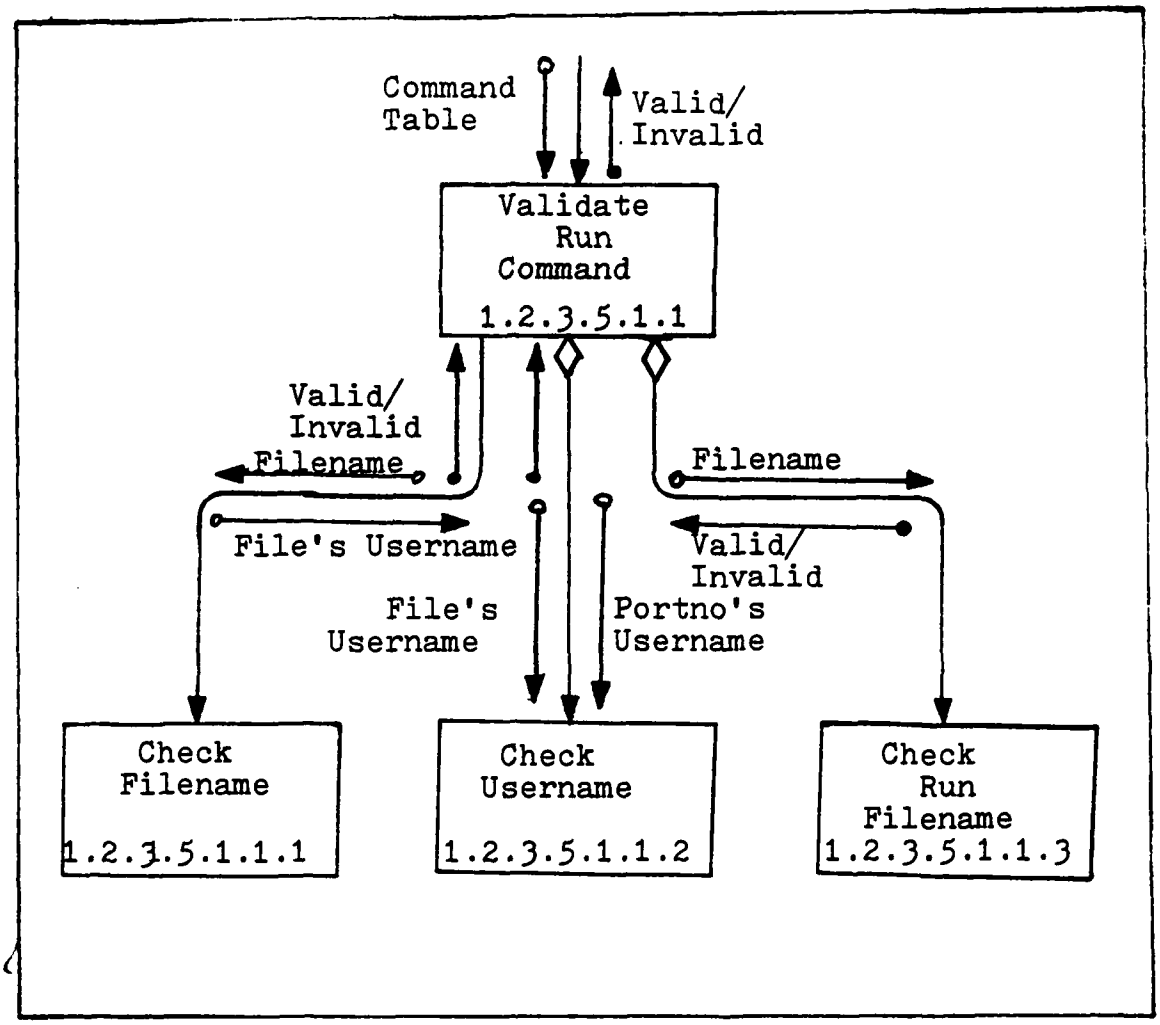
Note: (A) is a connector to the Build Error Message Module.

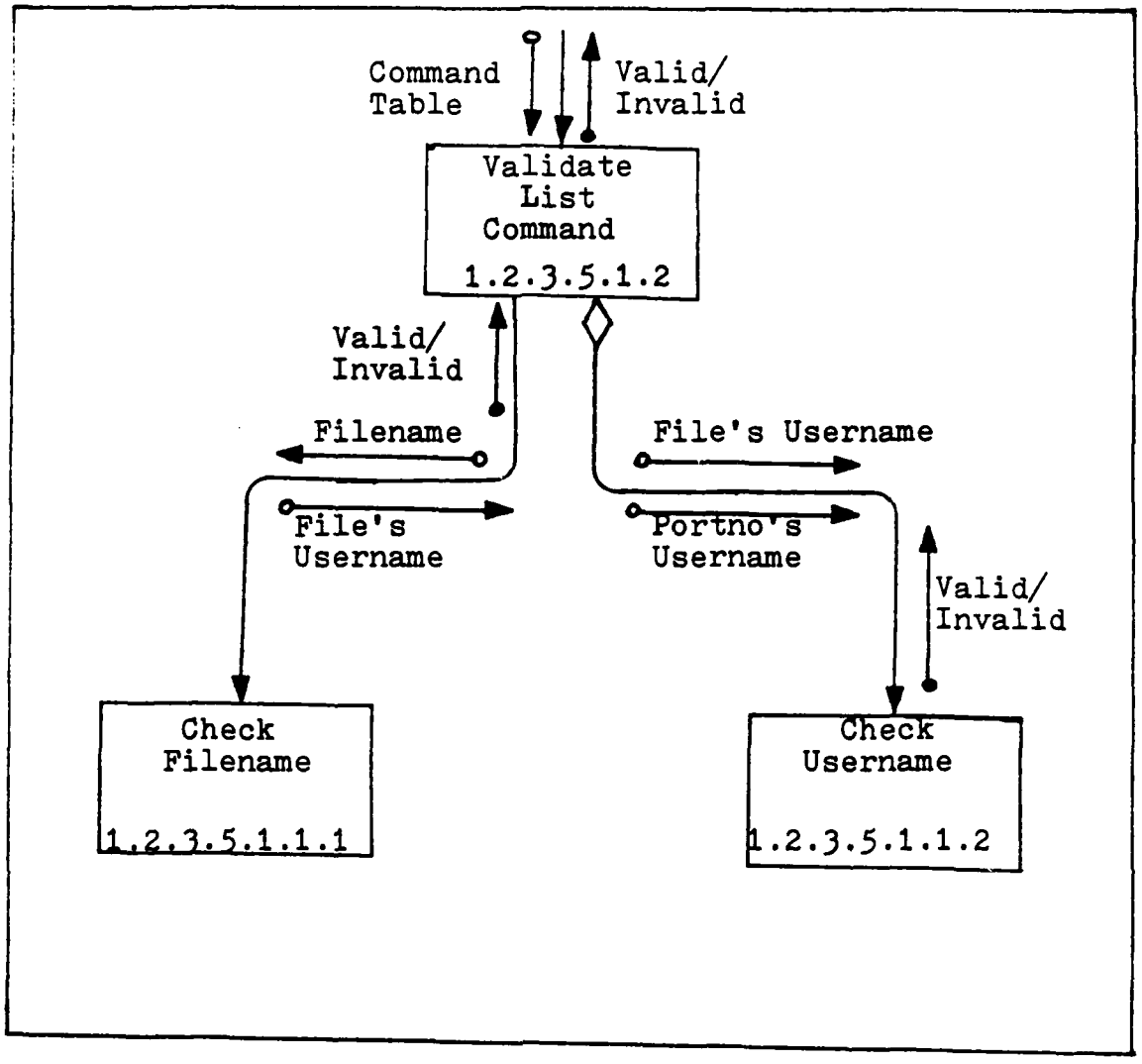

Portno
Log-in User
1.2.3.1
Portno
Logged-in
Set Up User Parameters
1.2.3.1.1
Build Message
1.2.3.1.2

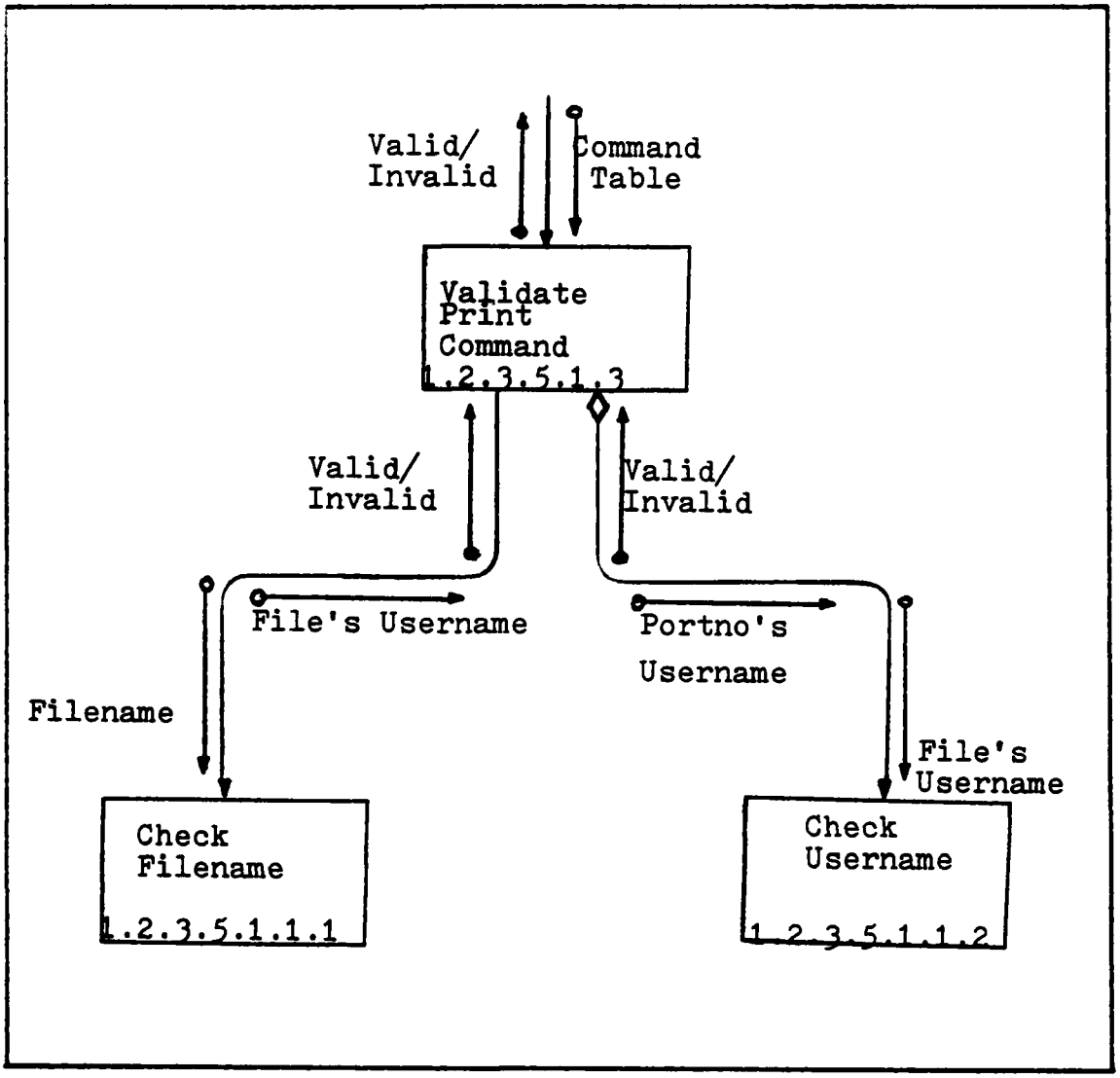

Portno
Set-Up User Parameters
1.2.3.1.1
Portno
Illegal User
Portno
Username & Password
Legal/ Illegal User
Build Message
1.2.3.1.2.
Read Command Line
1.2.2.2
Check User
1.2.3.1.1.1
Build Error Message
1.2.3.5.
1.1.1.3

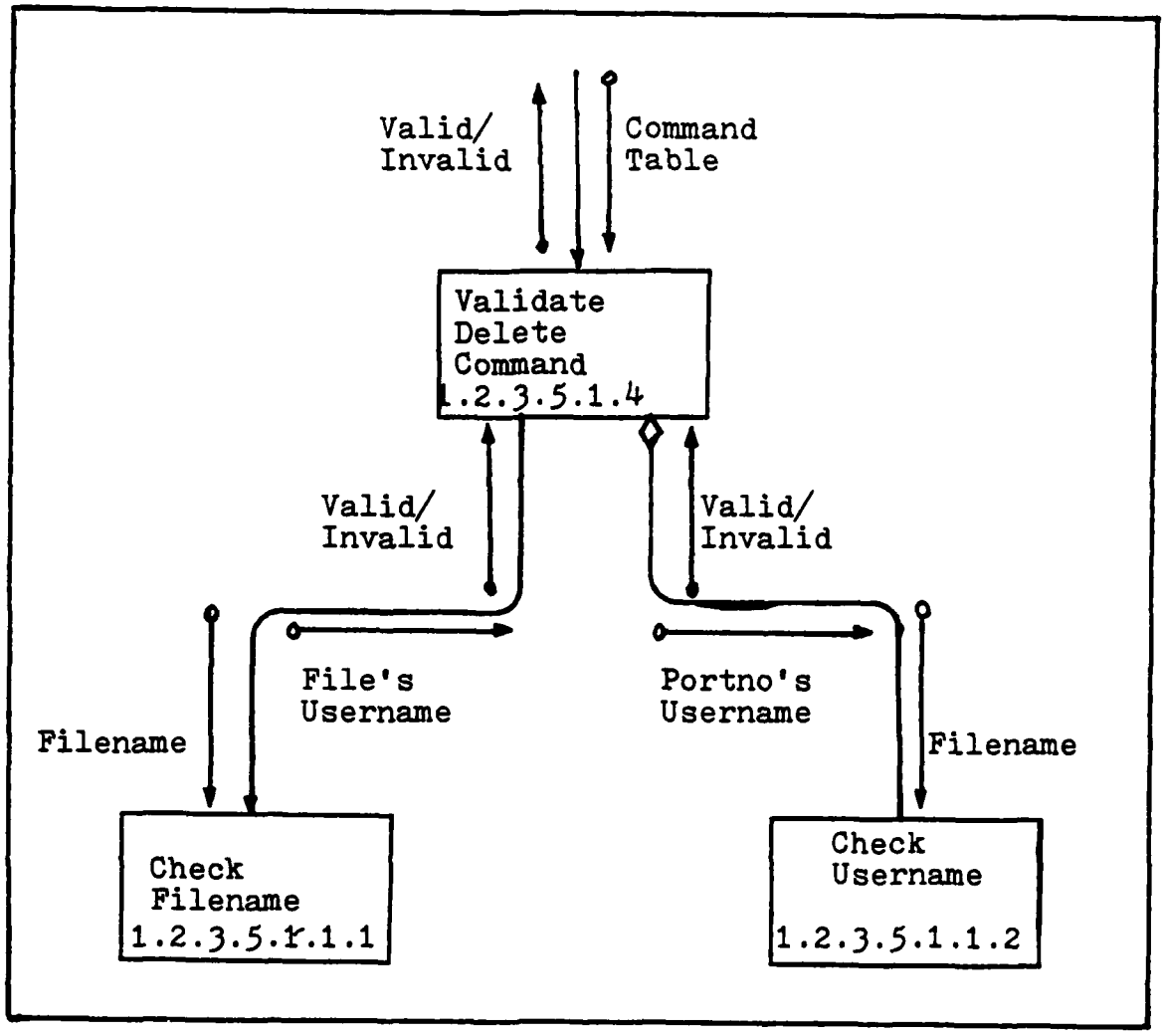

Command
Table
Execute
User
Command
1.2.3.5
Command
Table
Command
Table
Valid/Invalid
Validate
User
Command
1.2.3.5.1
Execute
Command
1.2.3.5.2

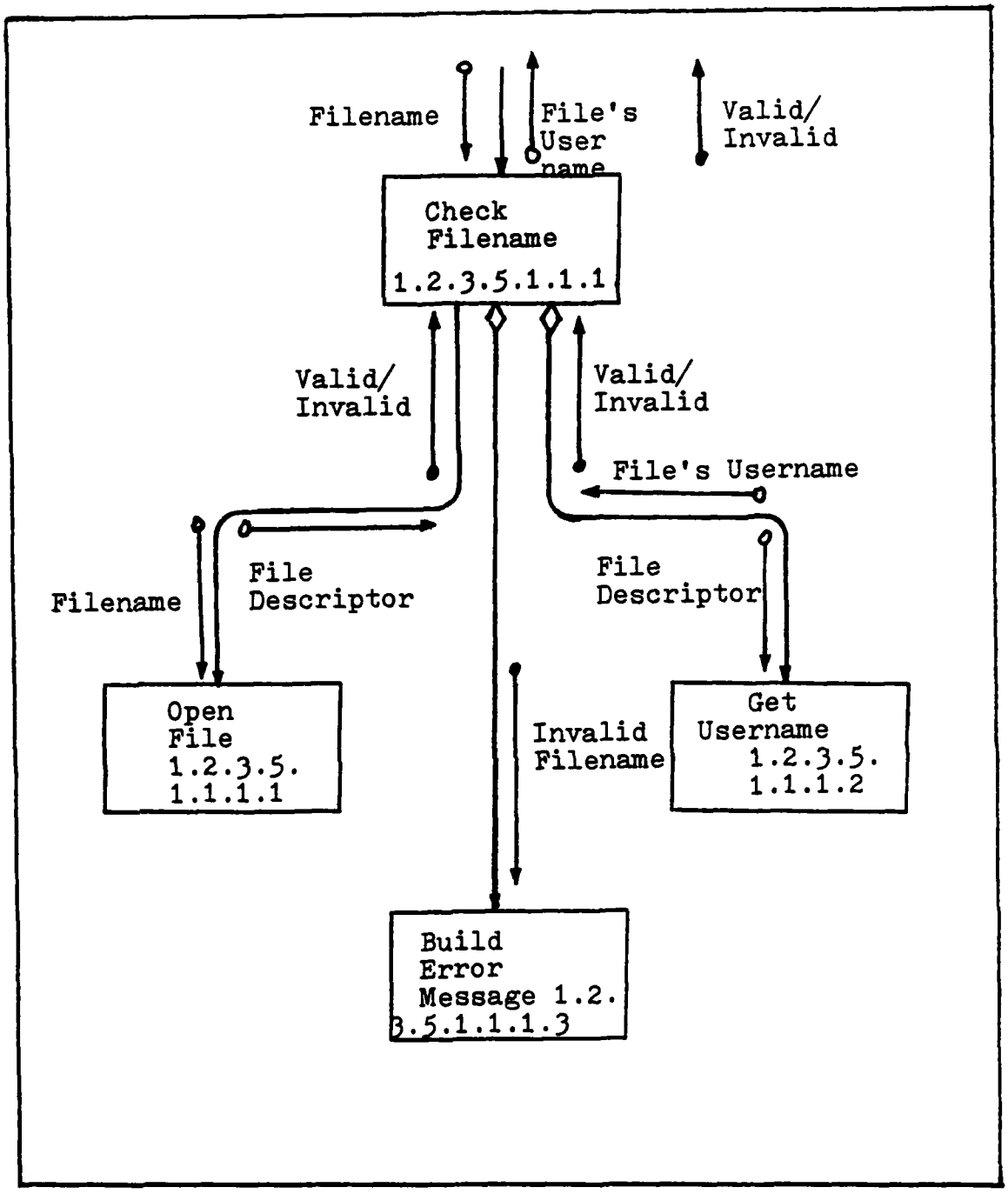

Command Table
Valid/ Invalid
Validate User Command 1.2.3.5.1
Command Table
Command Table
Valid/ Invalid
Command Table
Valid/ Invalid
Command Table
Valid/ Invalid
Validate Run Command 1.2.3.5.1.1
Validate List Command 1.2.3.5.1.2
Validate Print Command 1.2.3.5.1.3.
Validate Delete Command 1.2.3.5.1.4

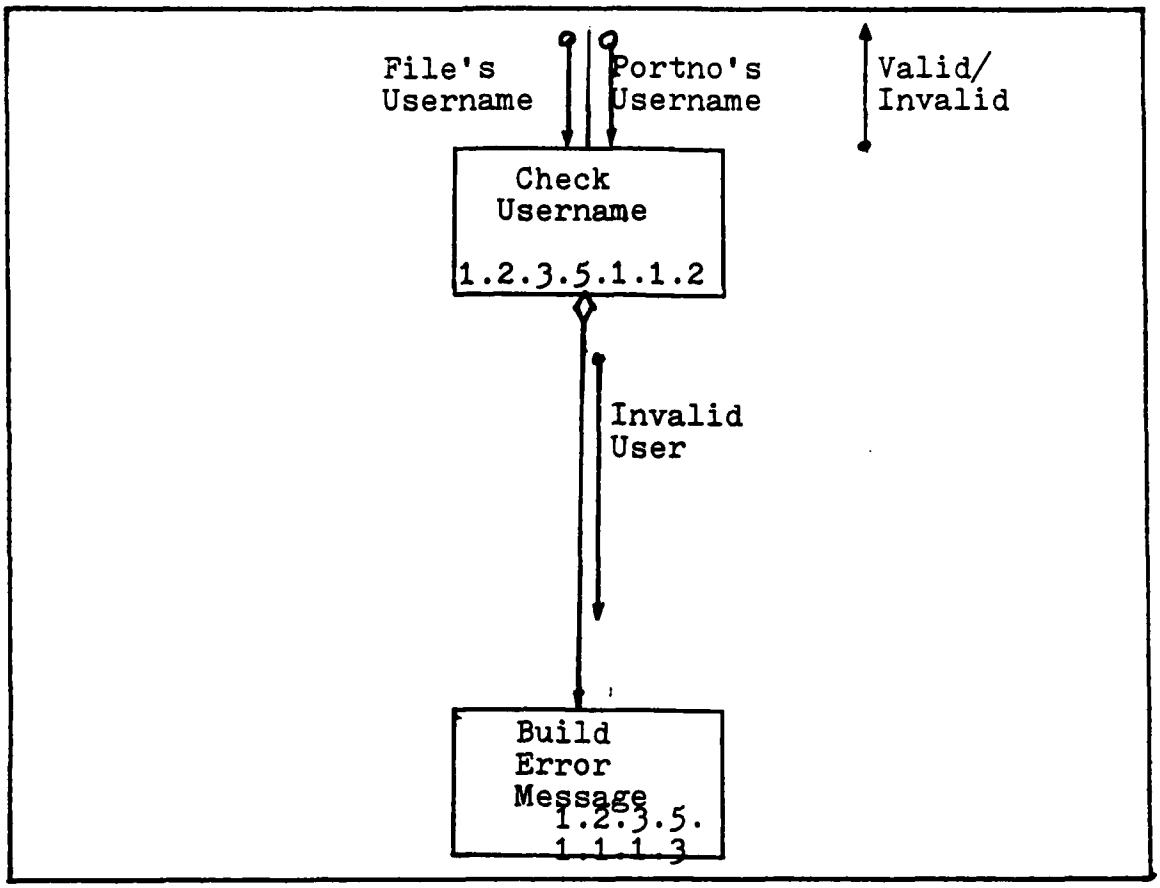

Command Table
Valid/ Invalid
Validate Run Command
1.2.3.5.1.1
Valid/ Invalid Filename
Filename
File's Username
Valid/ Invalid
File's Username
Portno's Username
Check Filename
1.2.3.5.1.1.1
Check Username
1.2.3.5.1.1.2
Check Run Filename
1.2.3.5.1.1.3

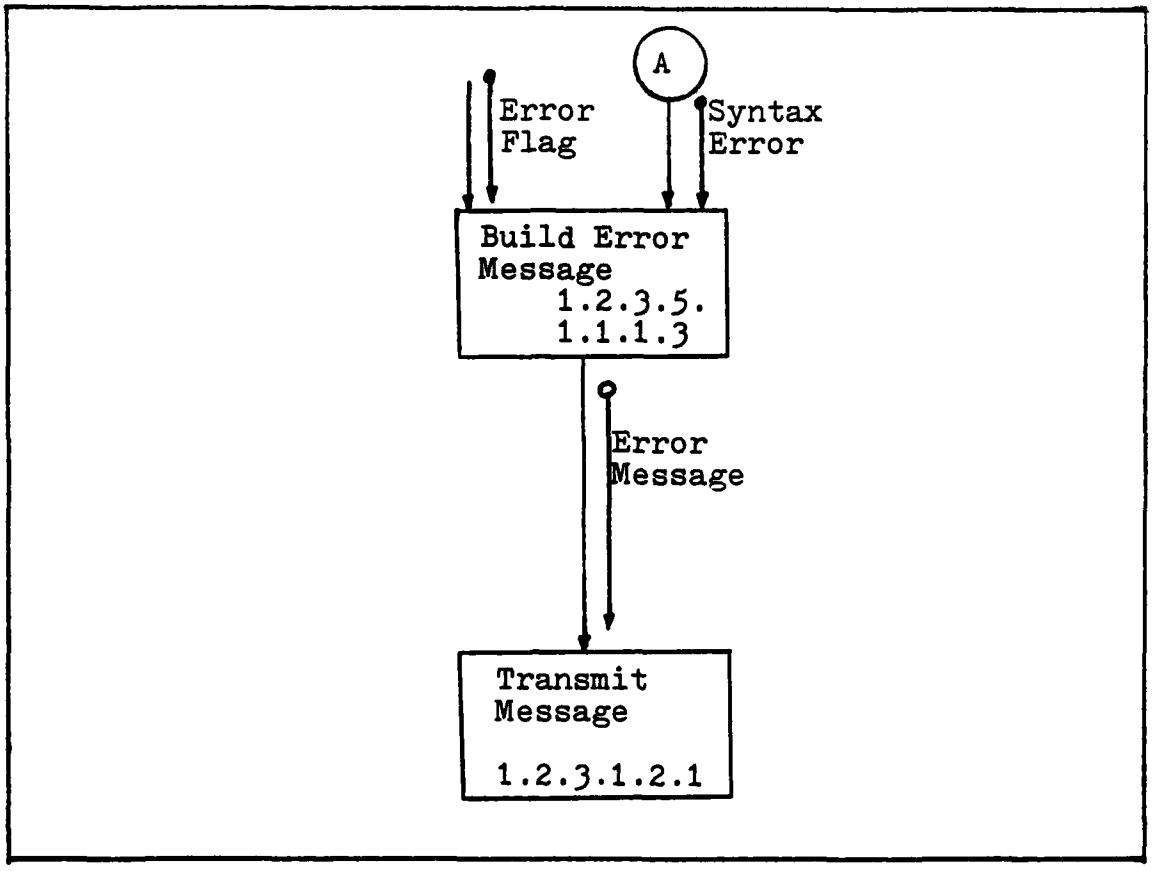

Command Table
Valid/ Invalid
Validate List Command
1.2.3.5.1.2
Valid/ Invalid
Filename
File's Username
File's Username
Portno's Username
Valid/ Invalid
Check Filename
1.2.3.5.1.1.1
Check Username
1.2.3.5.1.1.2

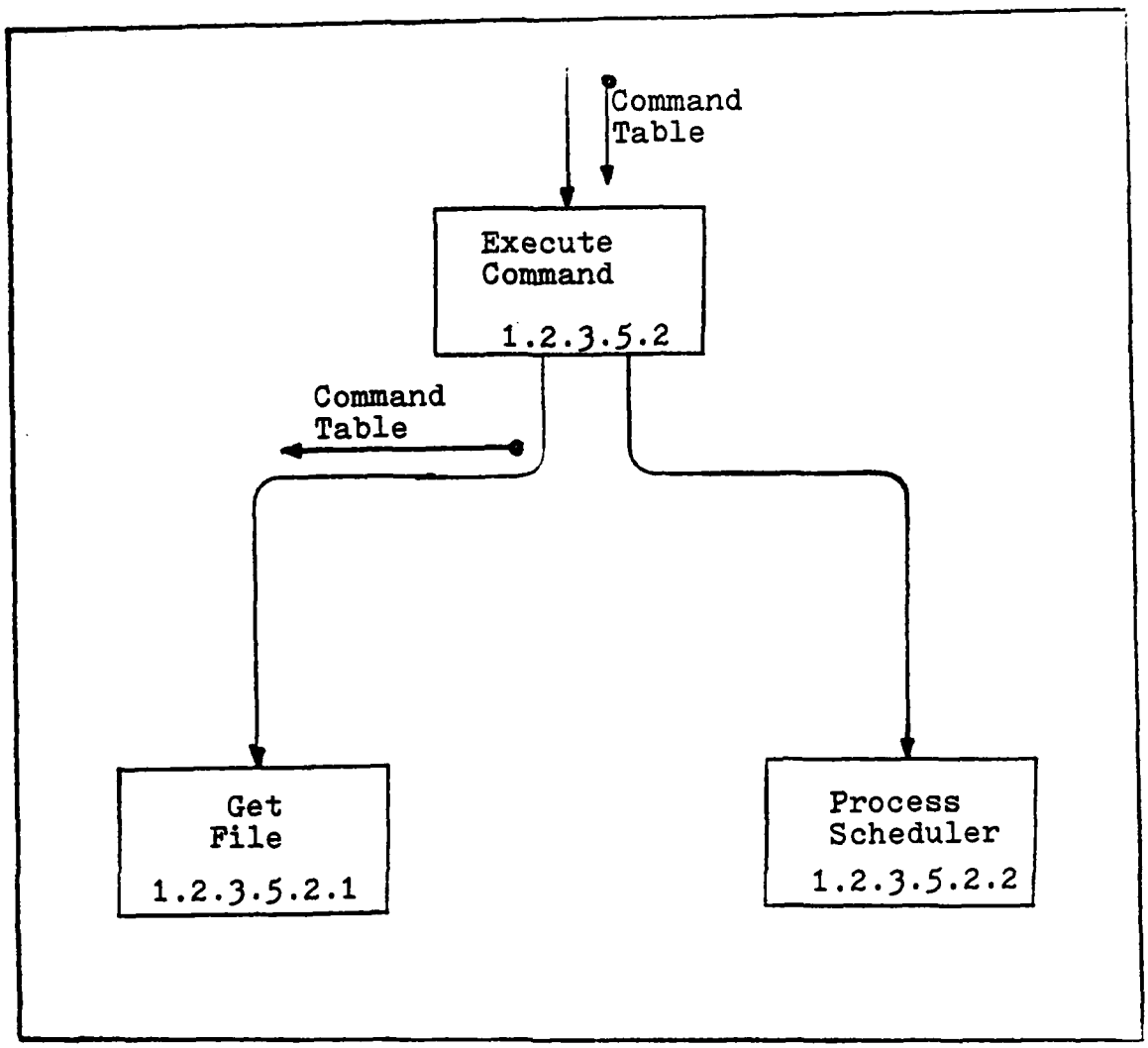

Valid/
Invalid
Command
Table
Validate
Print
Command
1.2.3.5.1.3
Valid/
Invalid
Valid/
Invalid
File's Username
Portno's
Username
Filename
File's
Username
Check
Filename
1.2.3.5.1.1.1
Check
Username
1.2.3.5.1.1.2

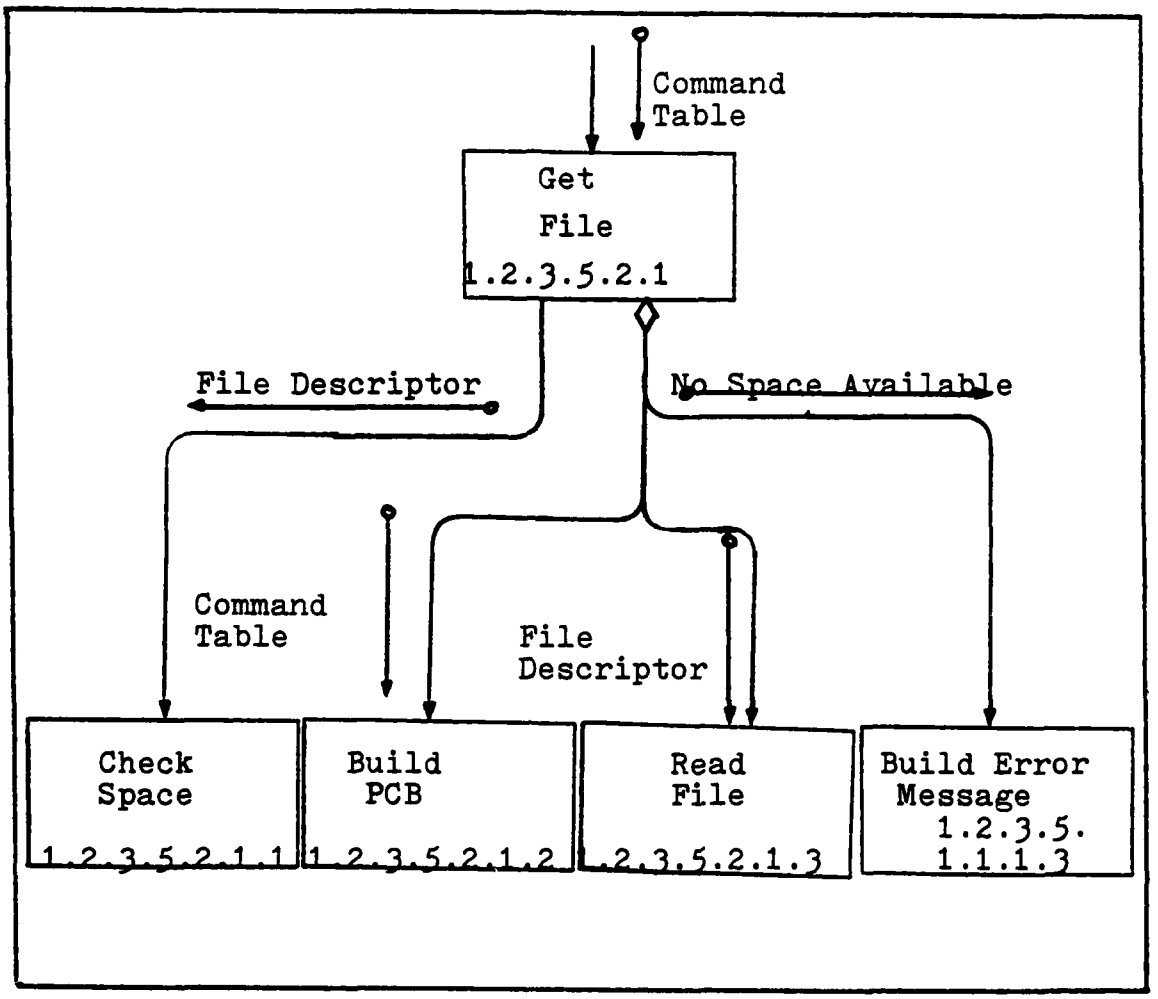

Valid/
Invalid
Command
Table
Validate
Delete
Command
1.2.3.5.1.4
Valid/
Invalid
Valid/
Invalid
File's
Username
Portno's
Username
Filename
Filename
Check
Filename
1.2.3.5.1.1.1
Check
Username
1.2.3.5.1.1.2

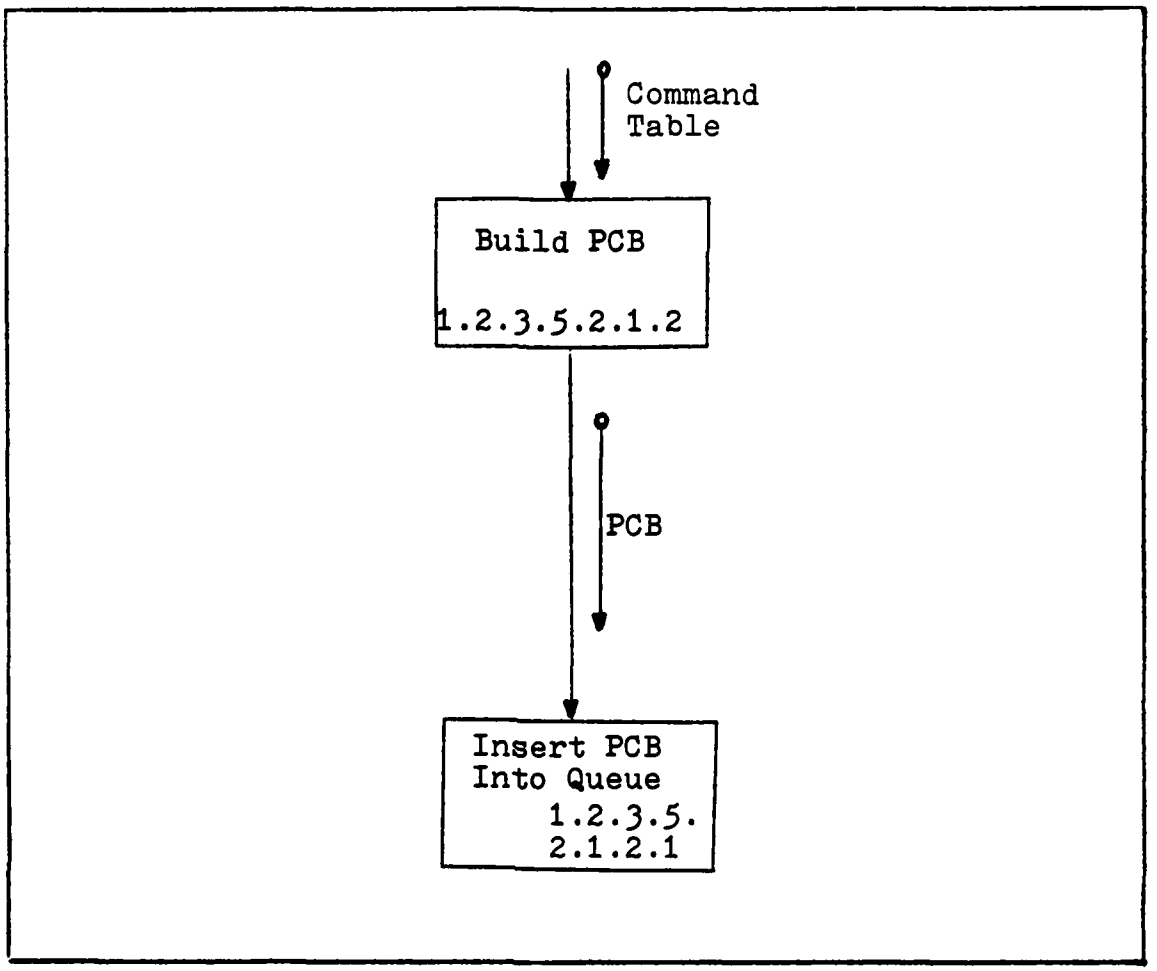

Filename
File's User name
Valid/ Invalid
Check Filename 1.2.3.5.1.1.1
Valid/ Invalid
Valid/ Invalid
File's Username
Filename
File Descriptor
File Descriptor
Open File 1.2.3.5. 1.1.1.1
Invalid Filename
Get Username 1.2.3.5. 1.1.1.2
Build Error Message 1.2. 3.5.1.1.1.3

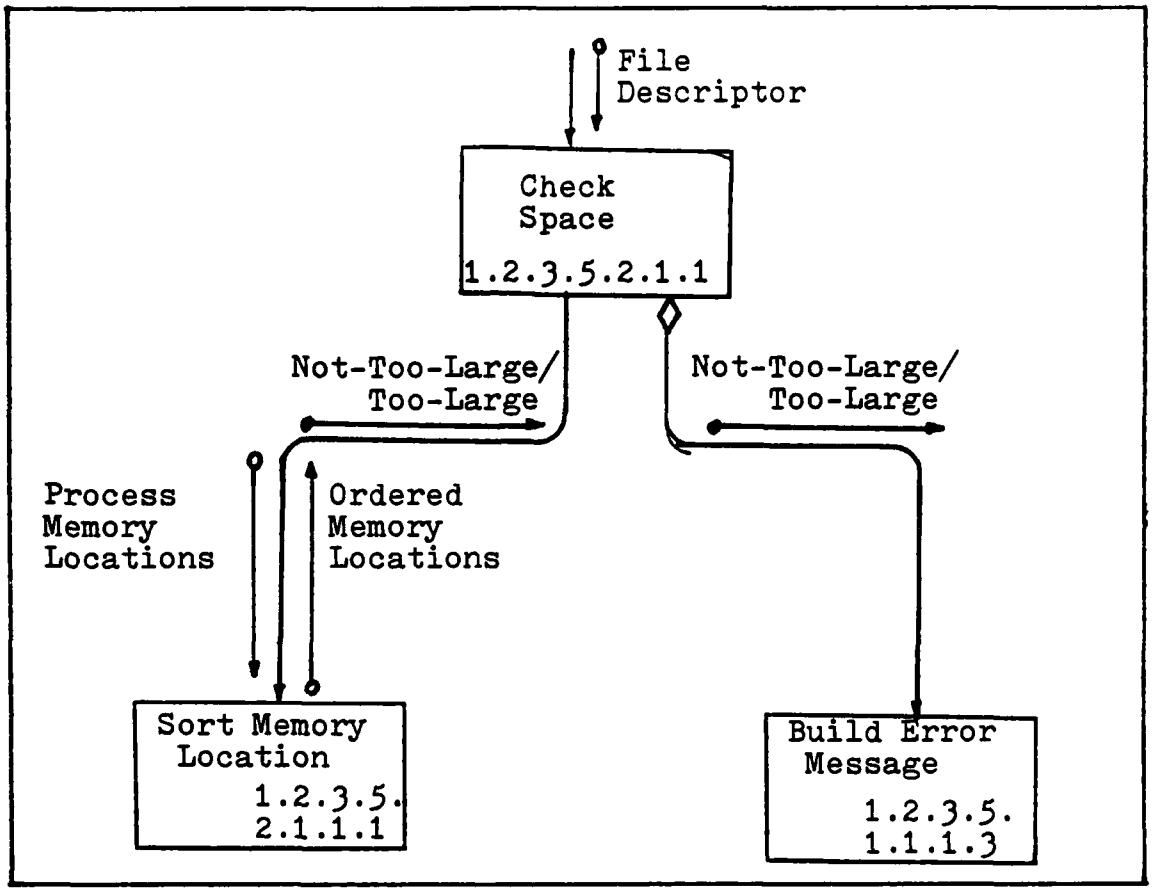

File's Username
Portno's Username
Valid/ Invalid
Check Username
1.2.3.5.1.1.2
Invalid User
Build Error Message
1.2.3.5.1.1.3

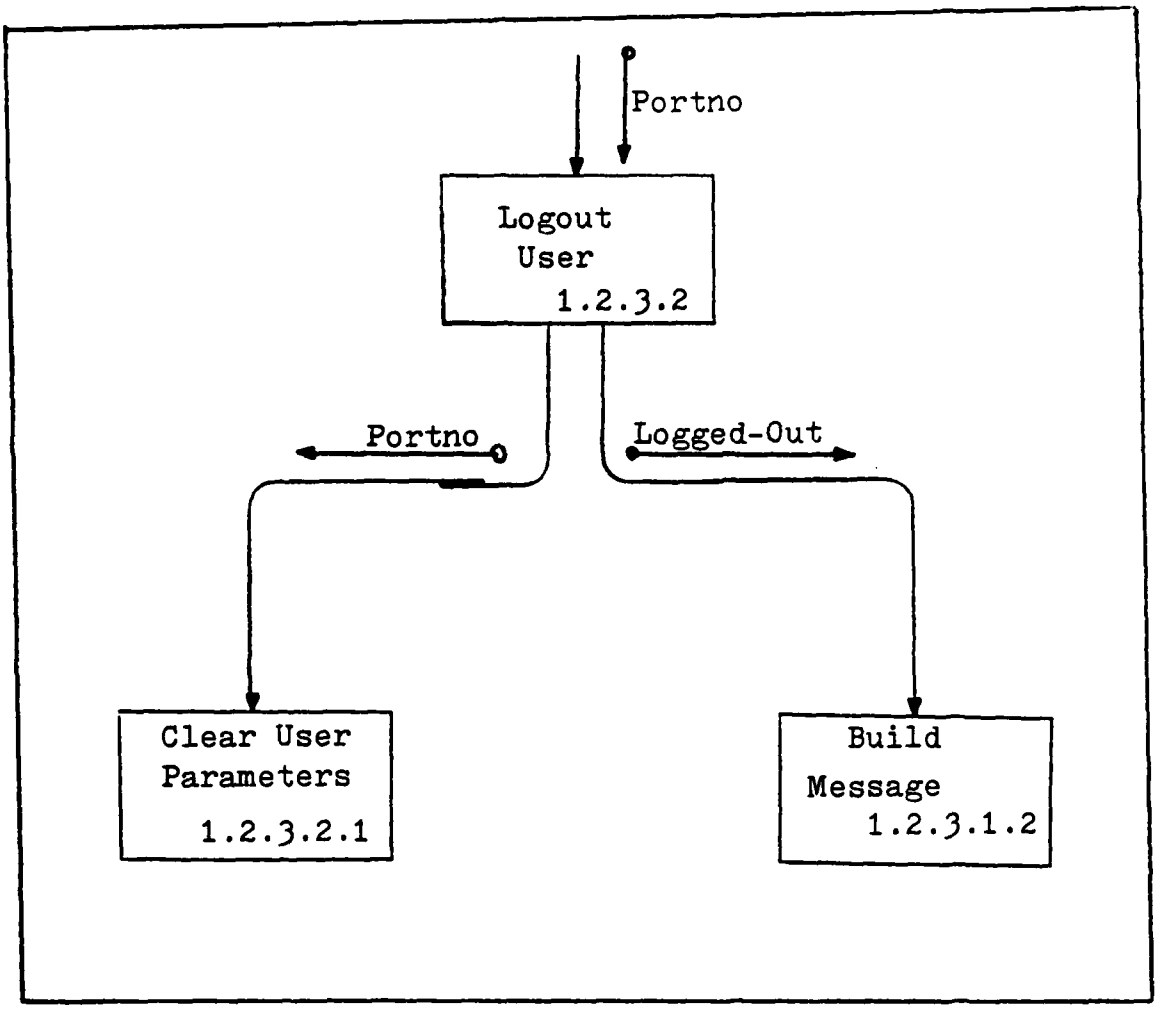

A
Error
Flag
Syntax
Error
Build Error
Message
1.2.3.5.
1.1.1.3
Error
Message
Transmit
Message
1.2.3.1.2.1

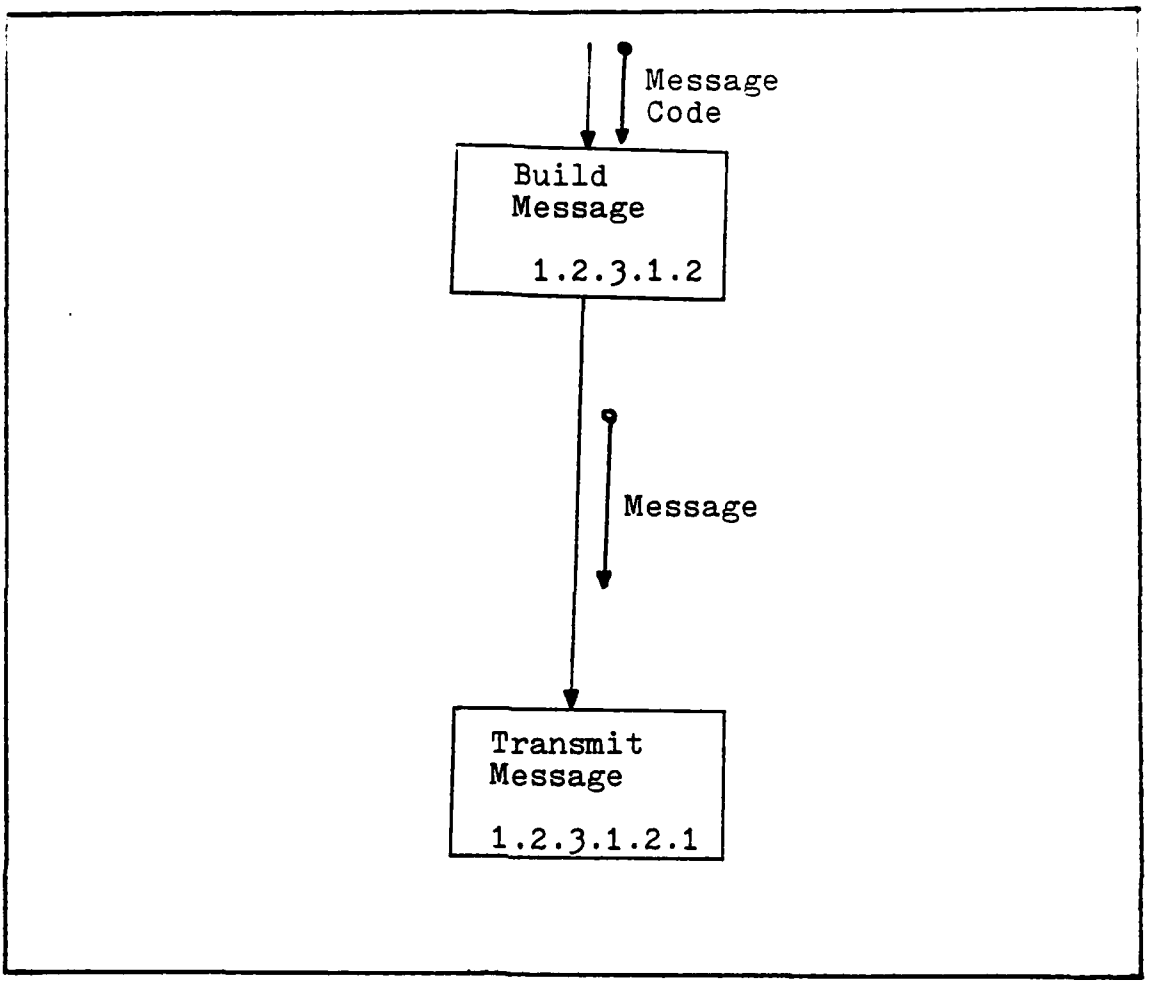

Command
Table
Execute
Command
1.2.3.5.2
Command
Table
Get
File
1.2.3.5.2.1
Process
Scheduler
1.2.3.5.2.2

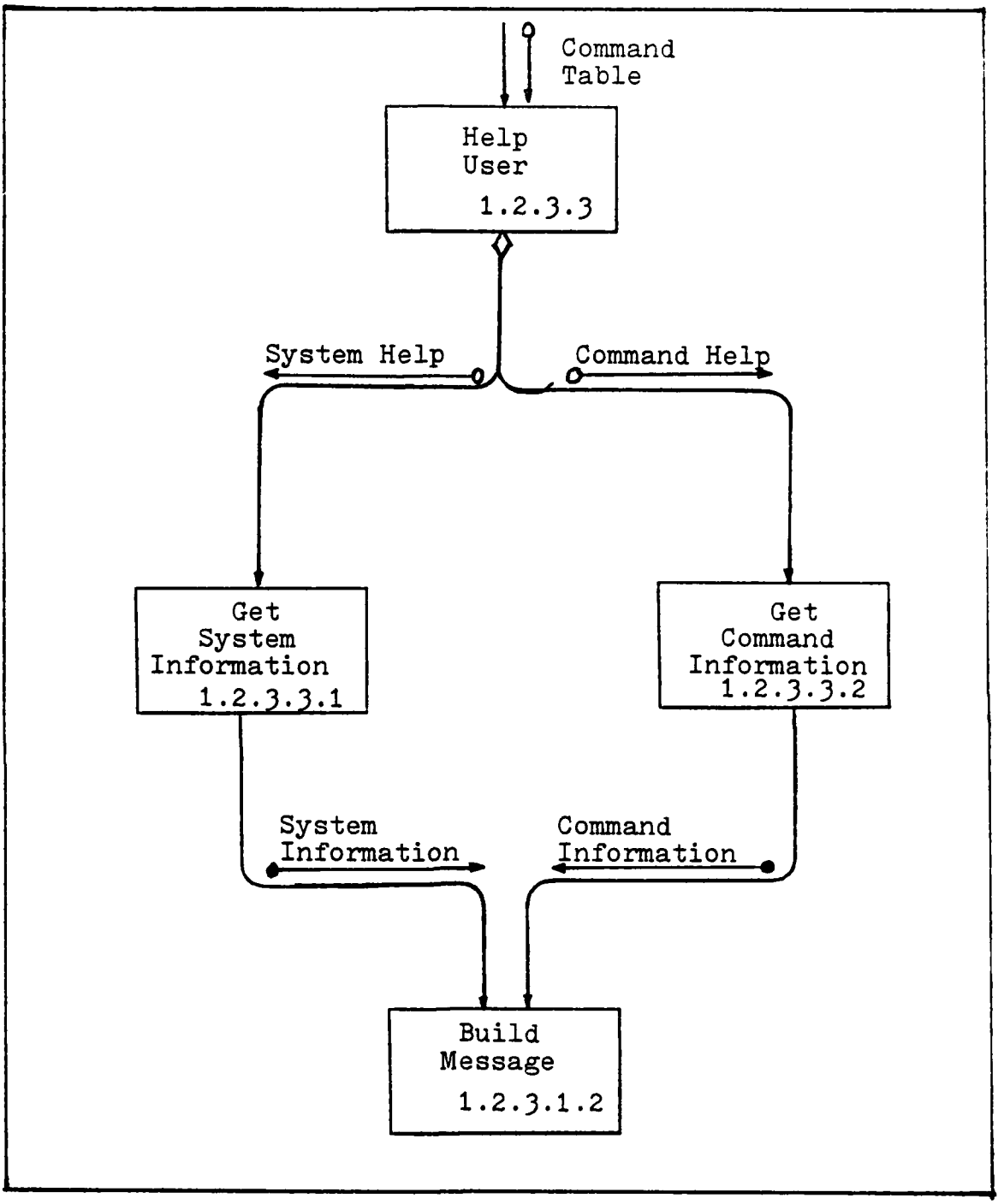

Command
Table
Get
File
1.2.3.5.2.1
File Descriptor
No Space Available
Command
Table
File
Descriptor
Check
Space
1.2.3.5.2.1.1
Build
PCB
1.2.3.5.2.1.2
Read
File
.2.3.5.2.1.3
Build Error
Message
1.2.3.5.
1.1.1.3

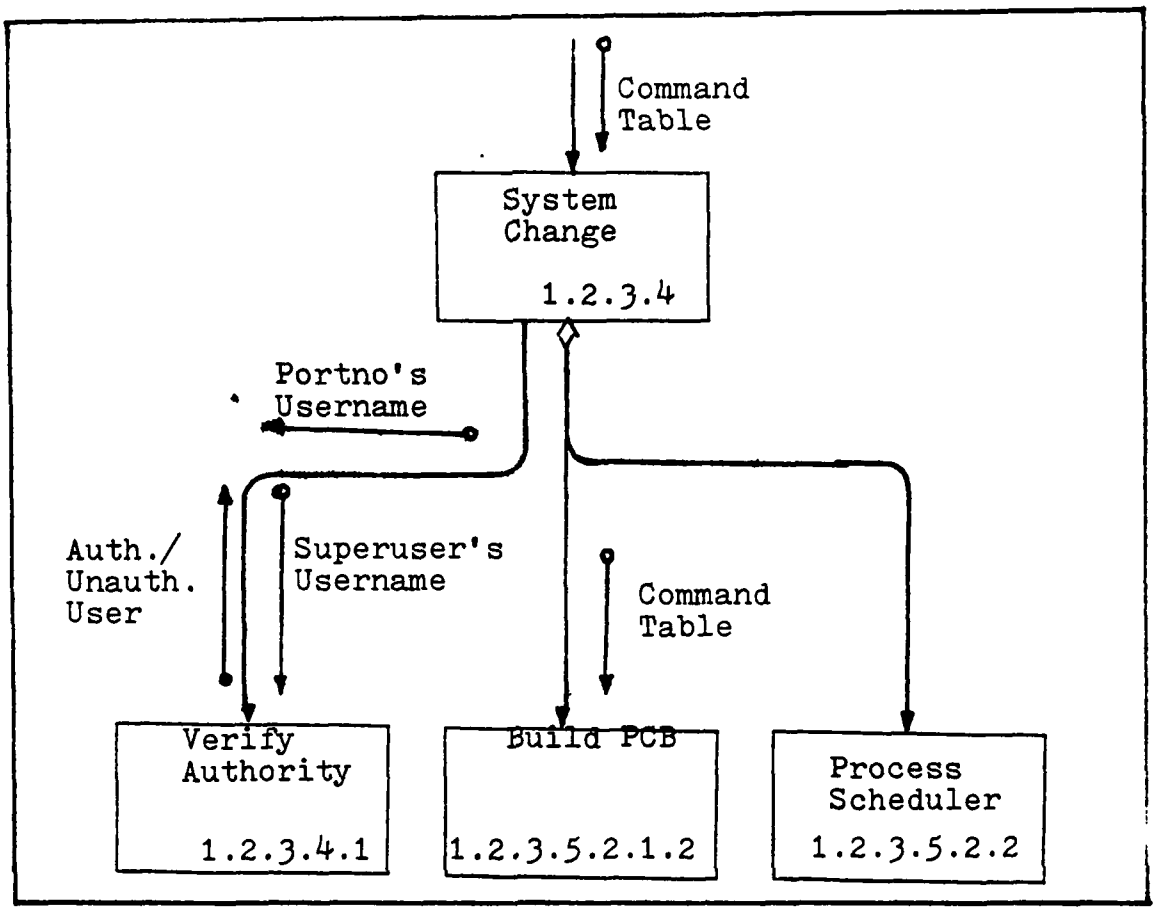

Command
Table
Build PCB
1.2.3.5.2.1.2
PCB
Insert PCB
Into Queue
1.2.3.5.
2.1.2.1

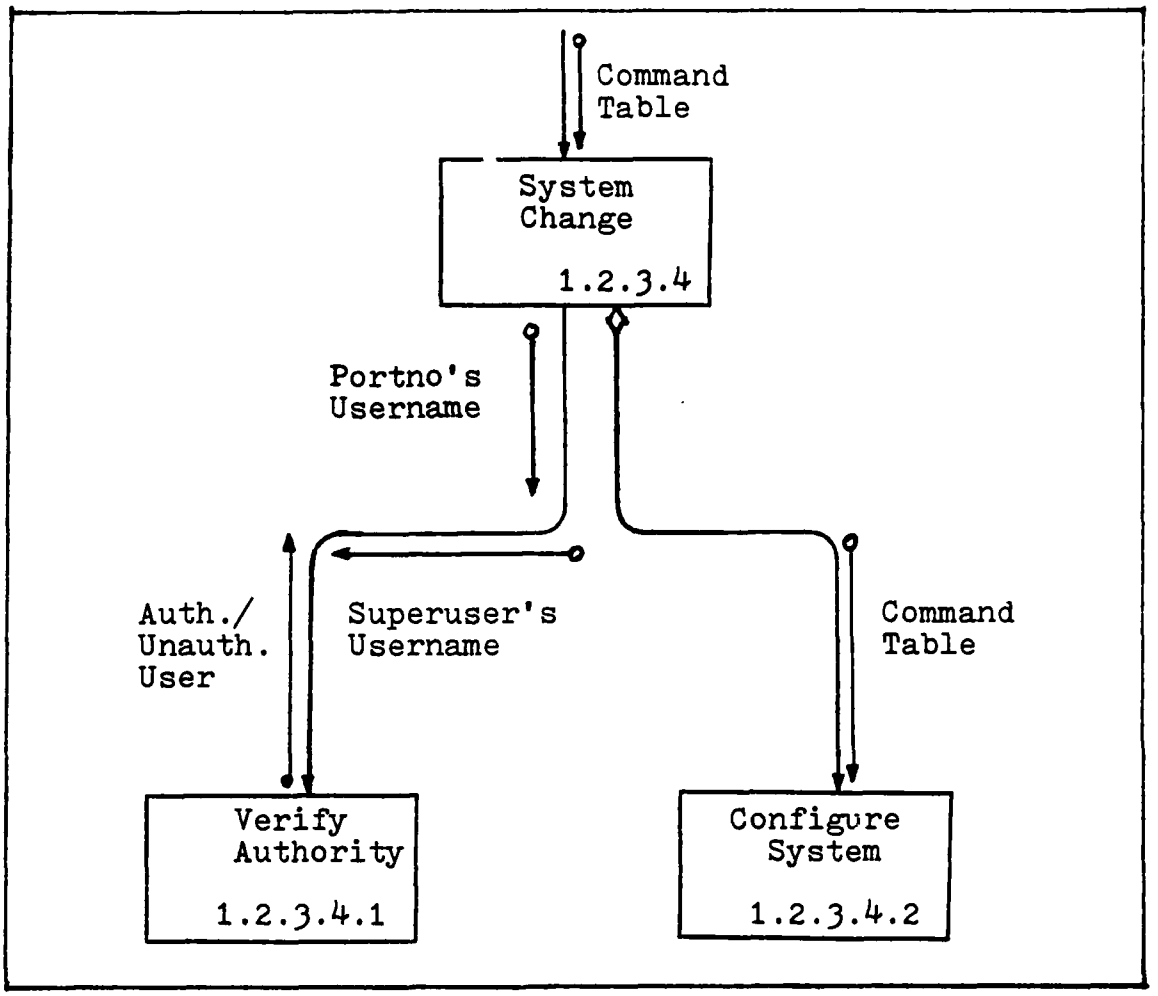

File
Descriptor
Check
Space
1.2.3.5.2.1.1
Not-Too-Large/
Too-Large
Not-Too-Large/
Too-Large
Process
Memory
Locations
Ordered
Memory
Locations
Sort Memory
Location
1.2.3.5.
2.1.1.1
Build Error
Message
1.2.3.5.
1.1.1.3

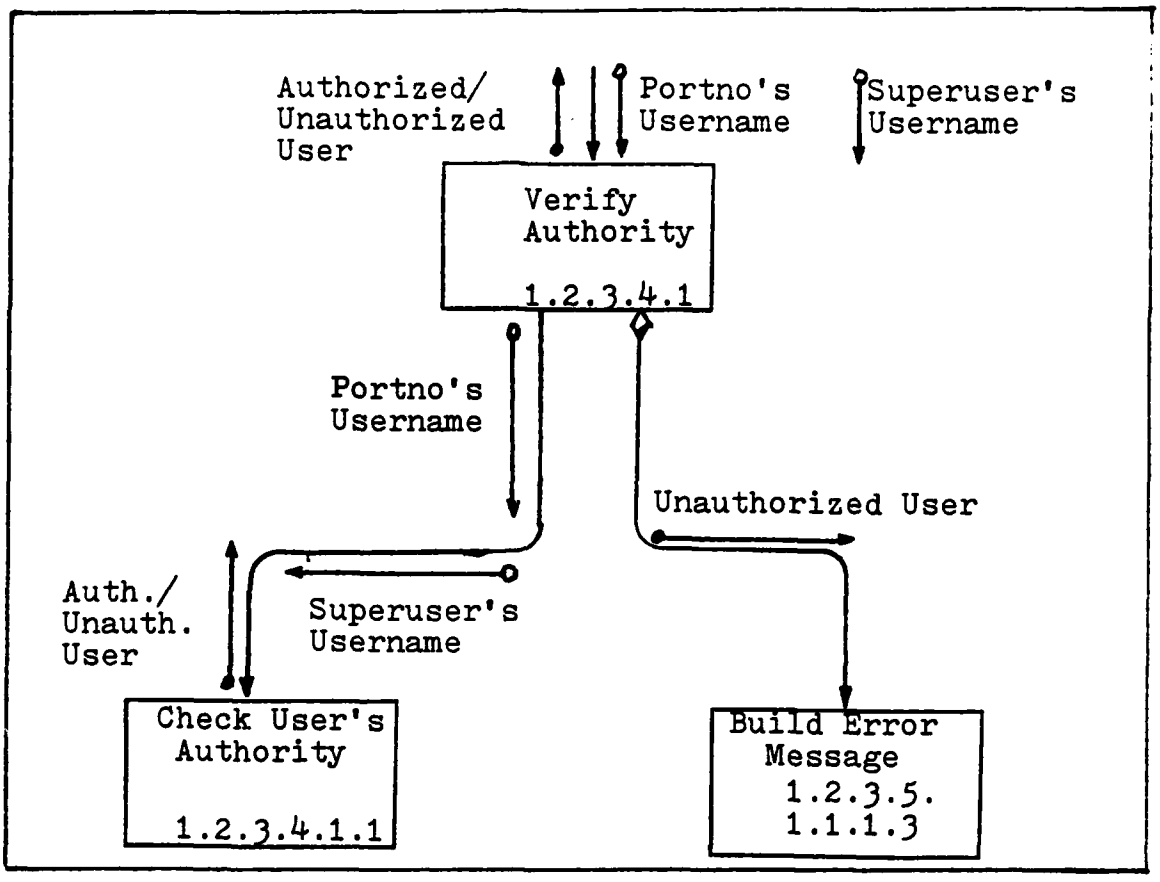

Portno
Logout
User
1.2.3.2
Portno
Logged-Out
Clear User
Parameters
1.2.3.2.1
Build
Message
1.2.3.1.2

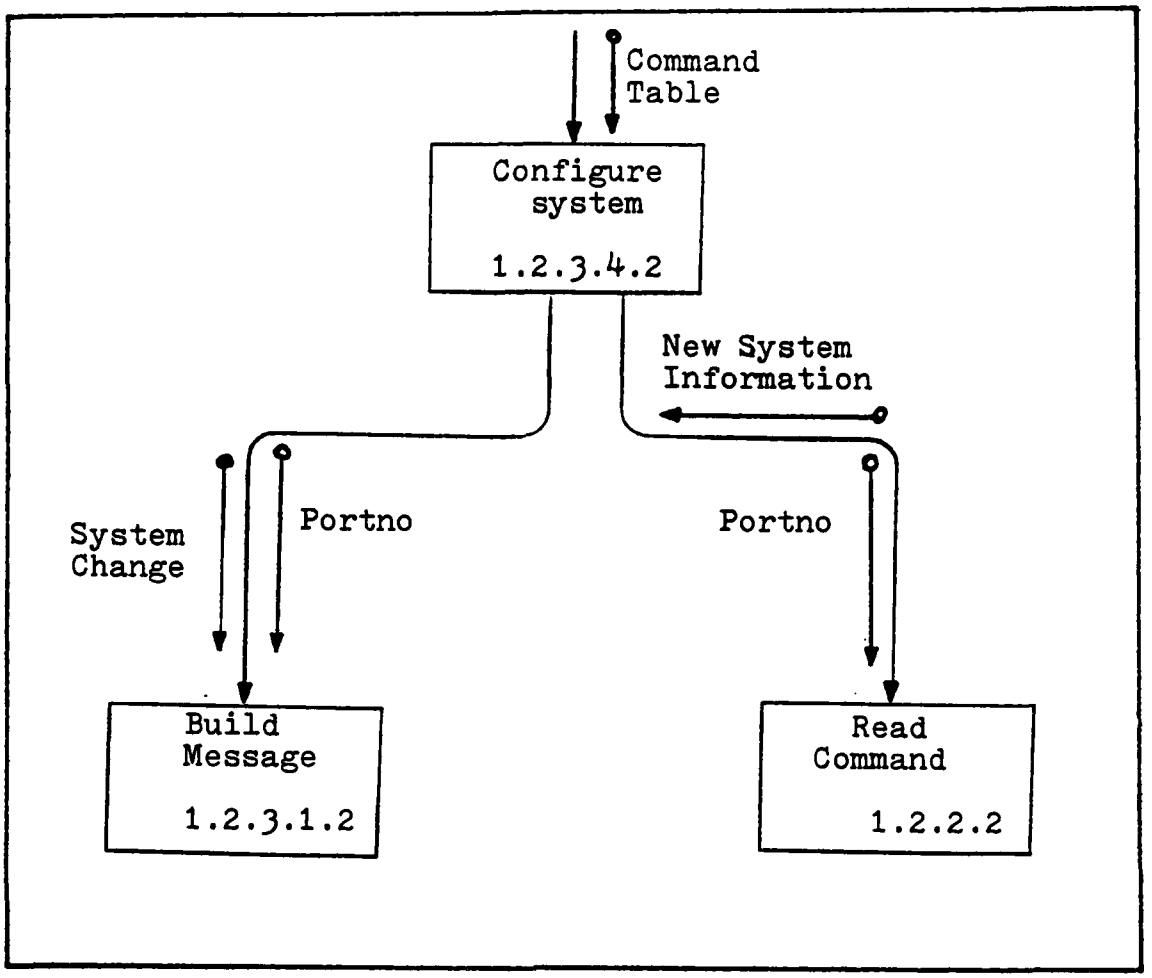

Message
Code
Build
Message
1.2.3.1.2
Message
Transmit
Message
1.2.3.1.2.1

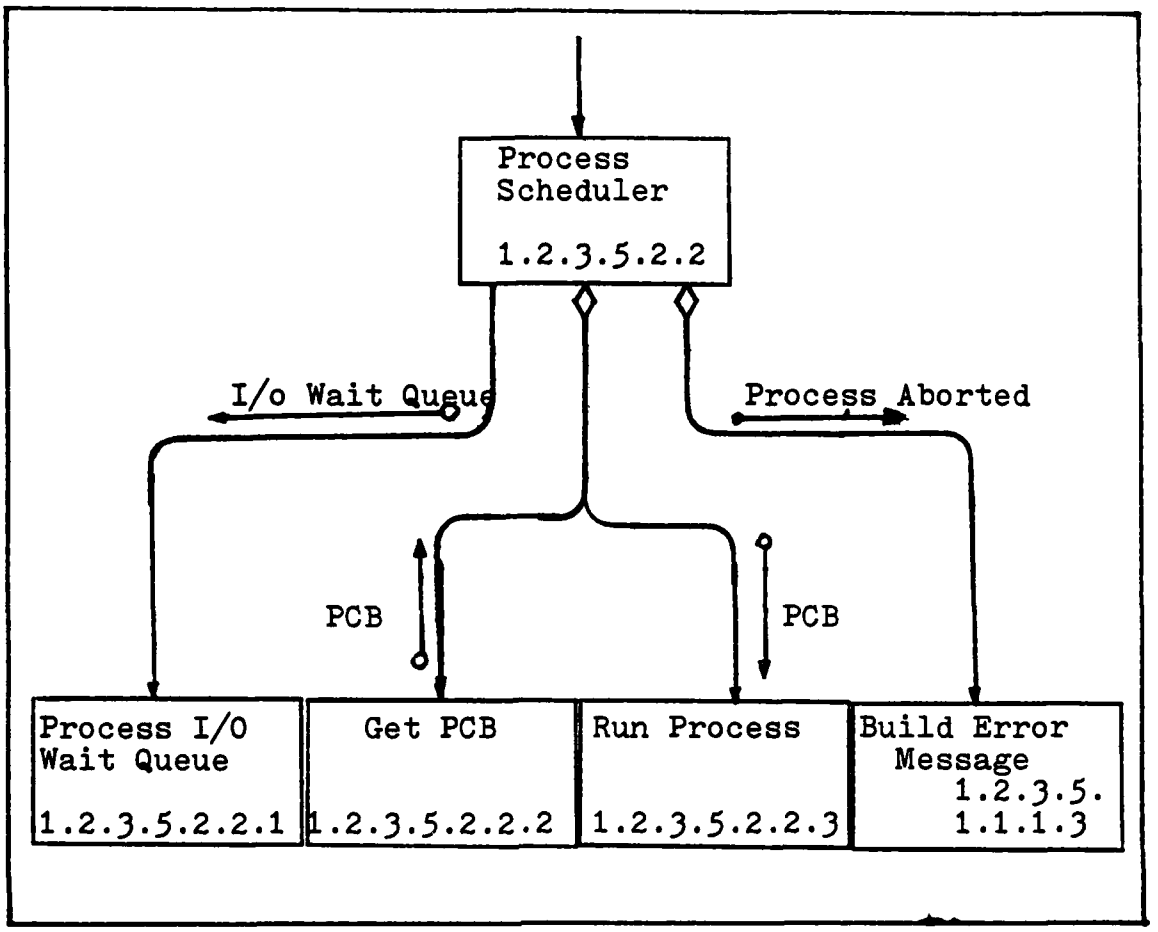

Command
Table
Help
User
1.2.3.3
System Help
Command Help
Get
System
Information
1.2.3.3.1
Get
Command
Information
1.2.3.3.2
System
Information
Command
Information
Build
Message
1.2.3.1.2

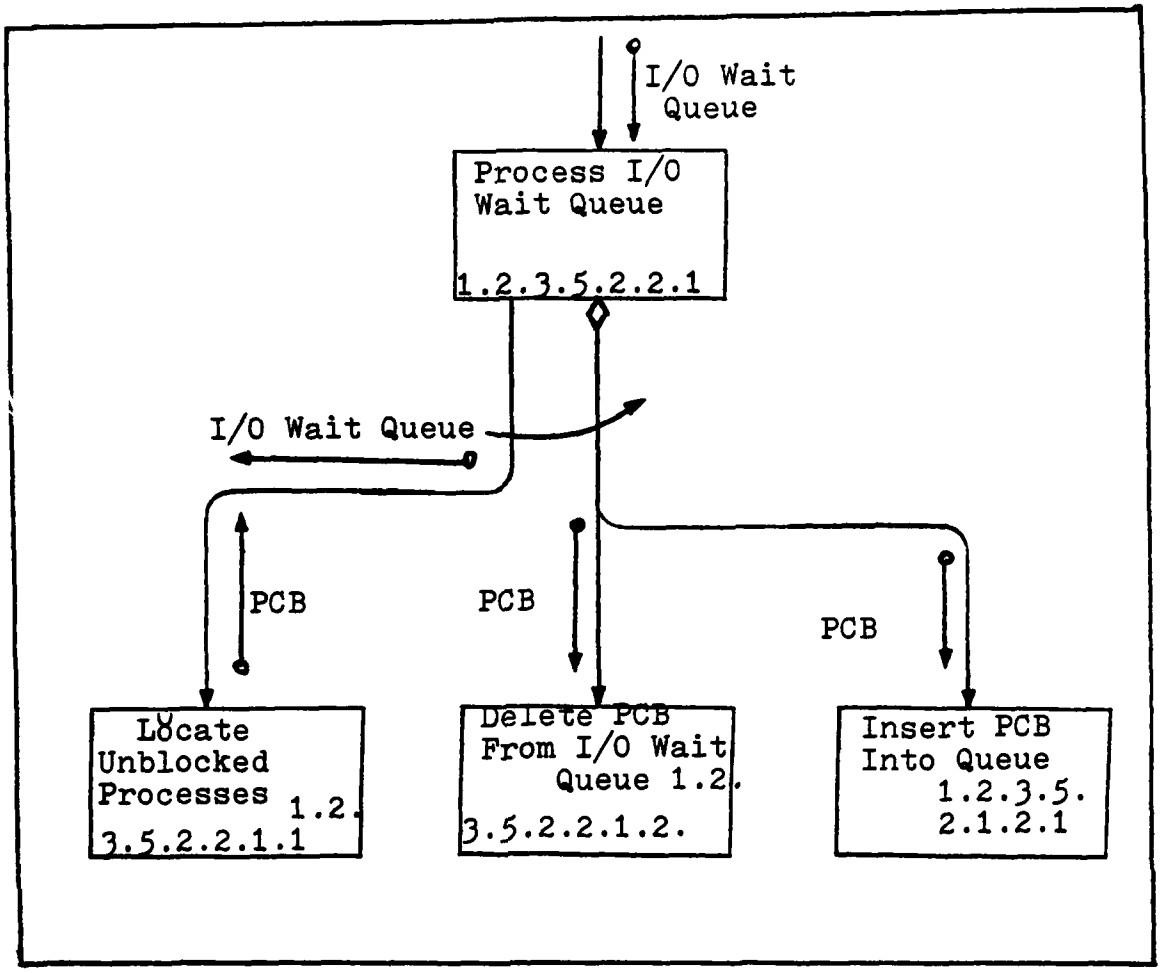

Command
Table
System
Change
1.2.3.4
Portno's
Username
Auth./
Unauth.
User
Superuser's
Username
Command
Table
Verify
Authority
1.2.3.4.1
Build PCB
1.2.3.5.2.1.2
Process
Scheduler
1.2.3.5.2.2

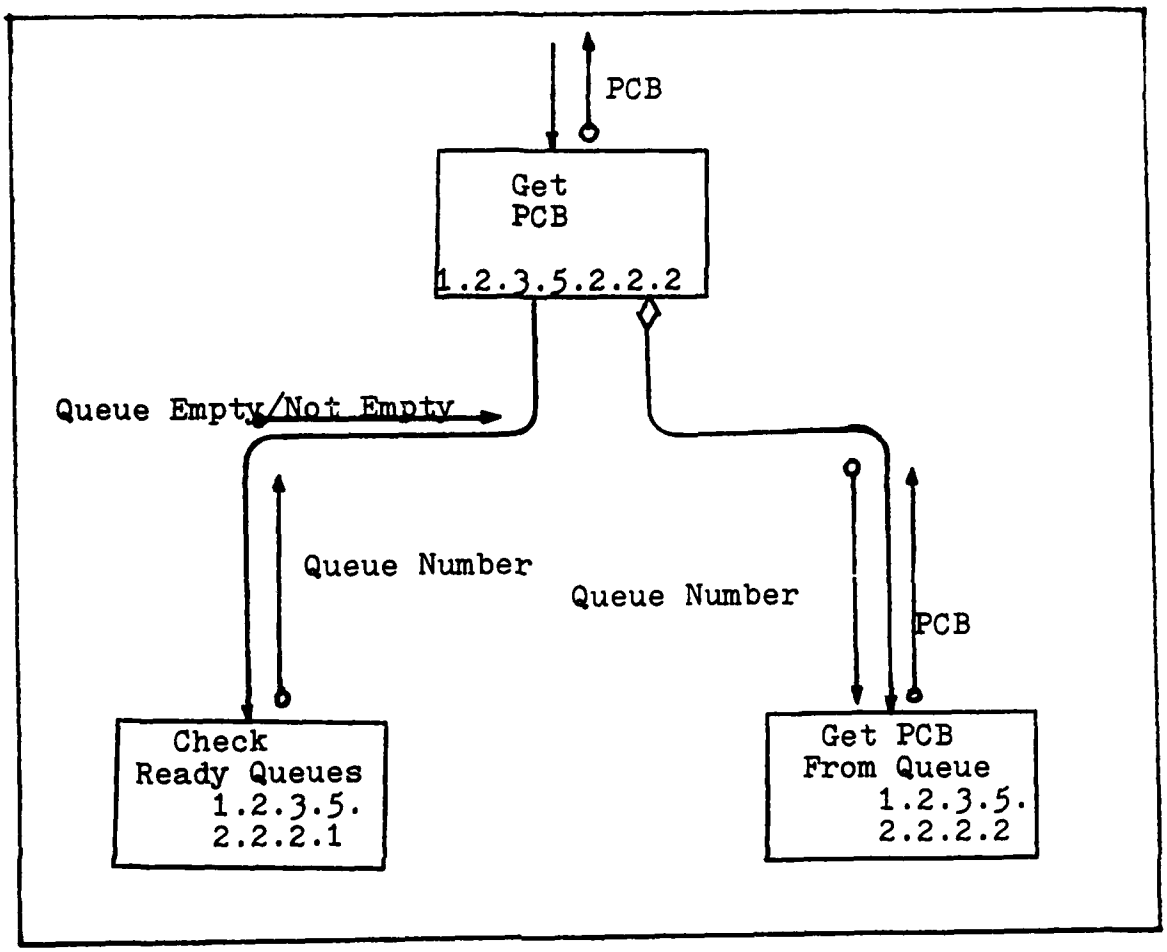

Command
Table
System
Change
1.2.3.4
Portno's
Username
Auth./
Unauth.
User
Superuser's
Username
Command
Table
Verify
Authority
1.2.3.4.1
Configure
System
1.2.3.4.2

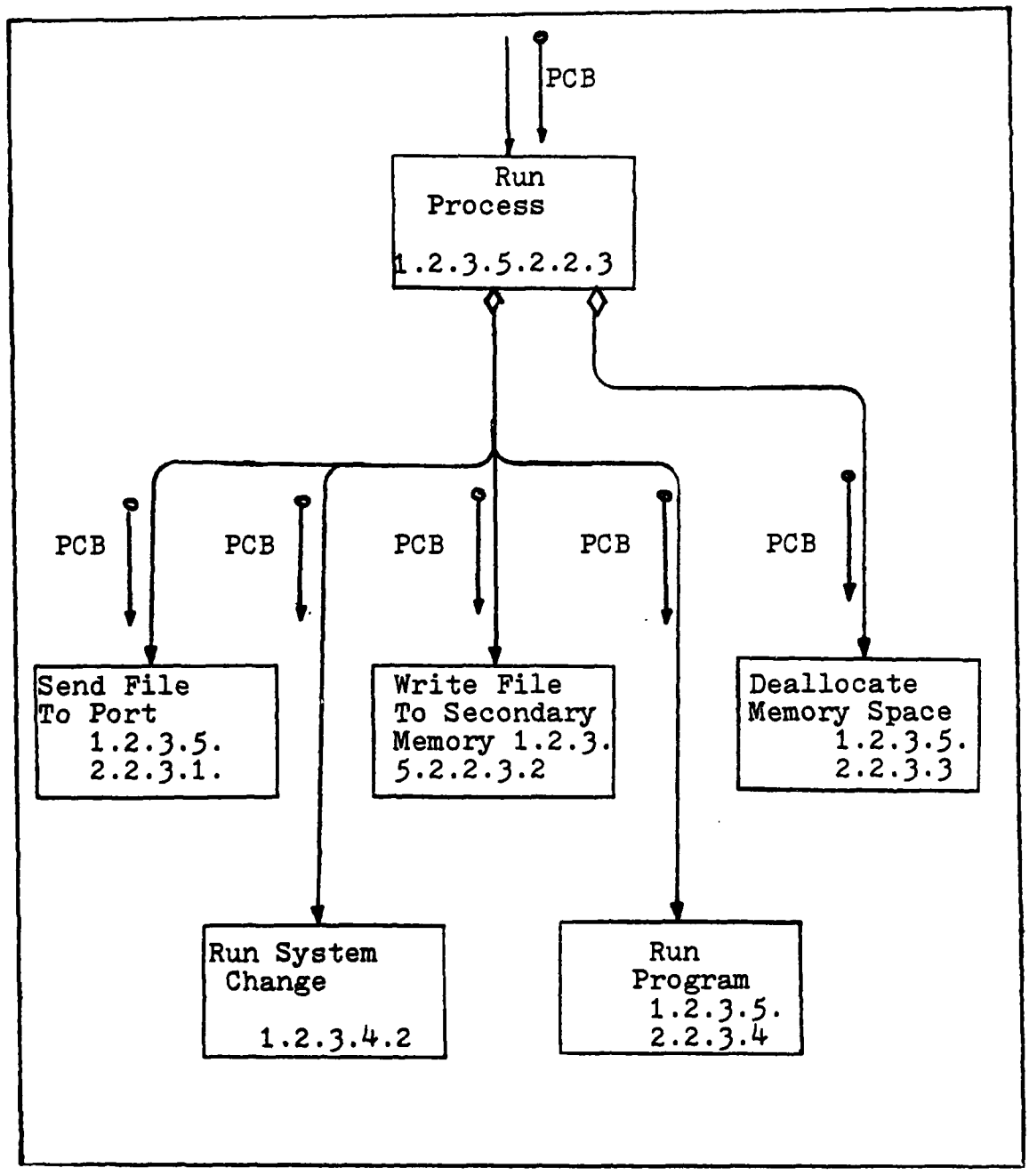

Authorized/
Unauthorized
User
Portno's
Username
Superuser's
Username
Verify
Authority
1.2.3.4.1
Portno's
Username
Unauthorized User
Auth./
Unauth.
User
Superuser's
Username
Check User's
Authority
1.2.3.4.1.1
Build Error
Message
1.2.3.5.
1.1.1.3

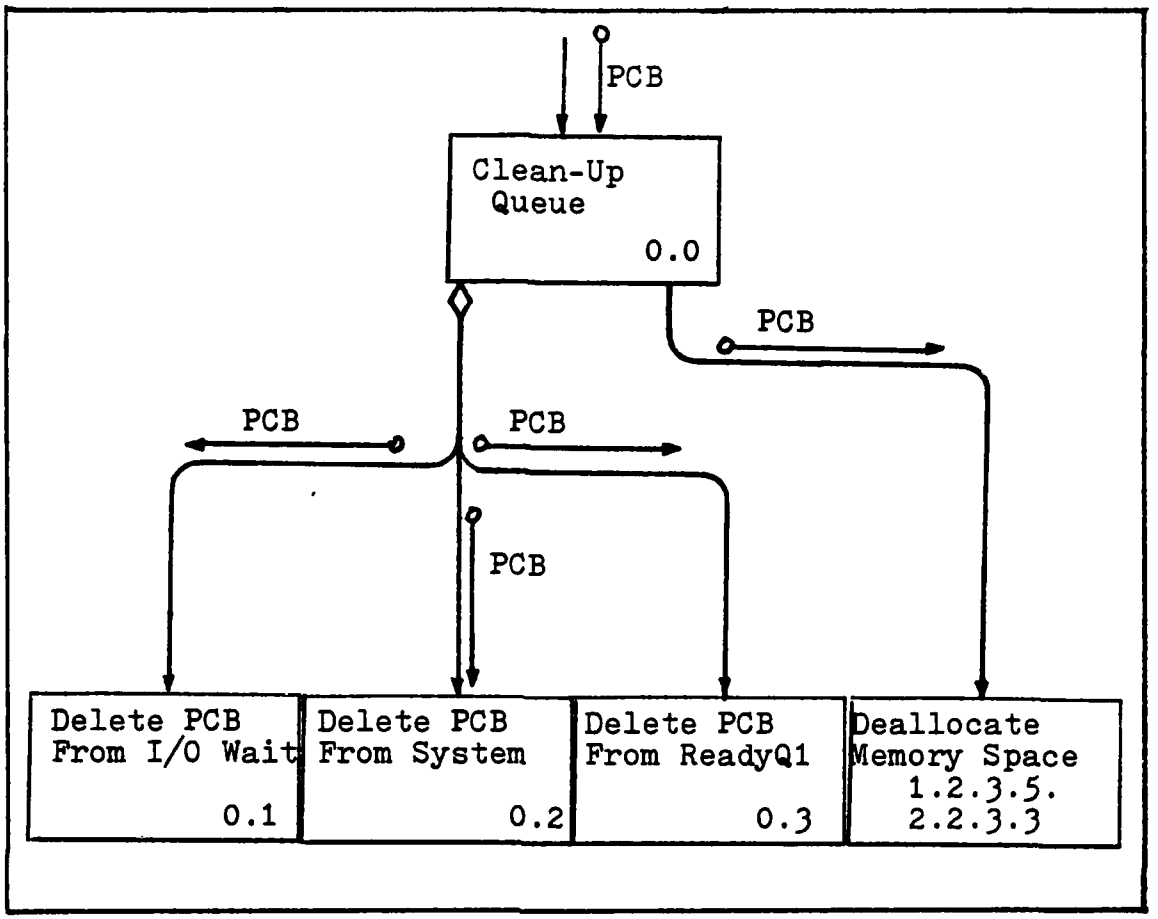

Command
Table
Configure
system
1.2.3.4.2
New System
Information
System
Change
Portno
Portno
Build
Message
1.2.3.1.2
Read
Command
1.2.2.2


Process
Scheduler
1.2.3.5.2.2
I/o Wait Queue
Process Aborted
PCB
PCB
Process I/O
Wait Queue
1.2.3.5.2.2.1
Get PCB
1.2.3.5.2.2.2
Run Process
1.2.3.5.2.2.3
Build Error
Message
1.2.3.5.
1.1.1.3


I/O Wait
Queue
Process I/O
Wait Queue
1.2.3.5.2.2.1
I/O Wait Queue
PCB
PCB
PCB
Locate
Unblocked
Processes 1.2.
3.5.2.2.1.1
Delete PCB
From I/O Wait
Queue 1.2.
3.5.2.2.1.2.
Insert PCB
Into Queue
1.2.3.5.
2.1.2.1


PCB
Get
PCB
1.2.3.5.2.2.2
Queue Empty/Not Empty
Queue Number
Queue Number
PCB
Check
Ready Queues
1.2.3.5.
2.2.2.1
Get PCB
From Queue
1.2.3.5.
2.2.2.2


PCB
Run
Process
1.2.3.5.2.2.3
PCB
PCB
PCB
PCB
PCB
Send File
To Port
1.2.3.5.
2.2.3.1.
Write File
To Secondary
Memory 1.2.3.
5.2.2.3.2
Deallocate
Memory Space
1.2.3.5.
2.2.3.3
Run System
Change
1.2.3.4.2
Run
Program
1.2.3.5.
2.2.3.4


PCB
Clean-Up
Queue
0.0
PCB
PCB
PCB
PCB
Delete PCB
From I/O Wait
0.1
Delete PCB
From System
0.2
Delete PCB
From ReadyQ1
0.3
Deallocate
Memory Space
1.2.3.5.
2.2.3.3

Appendix C

Process Description for AMOS

This appendix contains the process description of each structure module. The structure charts are located in Appendix B.

PROCESS NAME: Execute Bootstrap Program
PROCESS NUMBER: 1.0
PROCESS DESCRIPTION: This process loads the operating system from secondary memory into main memory. Upon completion of this the operating system is then executed.

PROCESS NAME: Load AMOS Into Memory
PROCESS NUMBER: 1.1
PROCESS DESCRIPTION: This process retrieves the operating system from secondary memory and places into main memory.

PROCESS NAME: Execute AMOS
PROCESS NUMBER: 1.2
PROCESS DESCRIPTION: This process executes the operating system that has already been loaded into main memory.

PROCESS NAME: Initialize Data Base
PROCESS NUMBER: 1.2.1
PROCESS DESCRIPTION: This process sets the initial values for specific data items used by the operating system.

PROCESS NAME: Parse Command Line
PROCESS NUMBER: 1.2.2
PROCESS DESCRIPTION: This process polls the on-line terminals, reads the next command line, and parses the command line into a command table. Process scheduler is called when terminals are idle.

PROCESS NAME: Determine Valid Command
PROCESS NUMBER: 1.2.3
PROCESS DESCRIPTION: This process determines the requested command and then checks for validity. If the command is found to be valid, it is then executed.

PROCESS NAME: Retrieve Data Base Information
PROCESS NUMBER: 1.2.1.1
PROCESS DESCRIPTION: This process retrieves the information that is used to initialize the operating system's variables.

PROCESS NAME: Initialize Variables
PROCESS NUMBER: 1.2.1.2
PROCESS DESCRIPTION: This process initializes the operating system's variables with the Data Base information.

PROCESS NAME: Poll Terminal Ports
PROCESS NUMBER: 1.2.2.1
PROCESS DESCRIPTION: This process uses a algorithm to poll the terminal ports to check for user requests.

PROCESS NAME: Read Command Line
PROCESS NUMBER: 1.2.2.2
PROCESS DESCRIPTION: This process reads the command line from the given port.

PROCESS NAME: Build Command Table
PROCESS NUMBER: 1.2.2.3
PROCESS DESCRIPTION: This process breaks the command line into separate parameters. These parameters are then placed into the command table.

PROCESS NAME: Log-in User
PROCESS NUMBER: 1.2.3.1
PROCESS DESCRIPTION: This process checks to see if the user is already logged-in. If found to be logged-in, then control is returned to calling module. Otherwise the user is attempted to be logged-in. The user is prompted for the username and password and reads the user's input. The username and password are verified. If they are valid, then the user block is initialized.

PROCESS NAME: Log-out User
PROCESS NUMBER: 1.2.3.2
PROCESS DESCRIPTION: This process clears the user block for the terminal which the log-out command originated.

PROCESS NAME: Help User
PROCESS NUMBER: 1.2.3.3
PROCESS DESCRIPTION: This process provides the user with the requested system or command information, if available.

PROCESS NAME: System Change
PROCESS NUMBER: 1.2.3.4
PROCESS DESCRIPTION: This process verifies that the user is the 'Superuser.' If found to be the 'Superuser,' then the system is reconfigured with the changes specified by the 'Superuser.'

PROCESS NAME: Execute User Command
PROCESS NUMBER: 1.2.3.5
PROCESS DESCRIPTION: This process determines if the specified user command is valid. If the command is valid, the command is then placed into main memory for execution.

PROCESS NAME: Set-up User Parameters
PROCESS NUMBER: 1.2.3.1.1
PROCESS DESCRIPTION: This process initializes the user block parameters.

PROCESS NAME: Build Message
PROCESS NUMBER: 1.2.3.1.2
PROCESS DESCRIPTION: This process constructs a message to be sent to a user and calls the Transmit Message module.

PROCESS NAME: Clear User Parameters
PROCESS NUMBER: 1.2.3.2.1
PROCESS DESCRIPTION: This process clears the user block parameters for the terminal which the log-out command originated.

PROCESS NAME: Get System Information
PROCESS NUMBER: 1.2.3.3.1
PROCESS DESCRIPTION: This process retrieves the requested system information and sends the information to the user.

PROCESS NAME: Get Command Information
PROCESS NUMBER: 1.2.3.3.2
PROCESS DESCRIPTION: This process retrieves the requested command information and sends the information to the user.

PROCESS NAME: Verify Authority
PROCESS NUMBER: 1.2.3.4.1
PROCESS DESCRIPTION: This process verifies that the user is the 'Superuser.'

PROCESS NAME: Configure System
PROCESS NUMBER: 1.2.3.4.2
PROCESS DESCRIPTION: This process configures the system's Data Base with the new information that is given by the 'Superuser.'

PROCESS NAME: Validate User Command
PROCESS NUMBER: 1.2.3.5.1
PROCESS DESCRIPTION: This process checks for validity of the specified user command (i.e. RUN, LIST, PRINT, DEL, and DIR).

PROCESS NAME: Execute Command
PROCESS NUMBER: 1.2.3.5.2
PROCESS DESCRIPTION: This process retrieves a file and calls the Process Scheduler module for execution.

PROCESS NAME: Check User
PROCESS NUMBER: 1.2.3.1.1.1
PROCESS DESCRIPTION: This process checks the user table to determine if the user is allowed system access.

PROCESS NAME: Transmit Message
PROCESS NUMBER: 1.2.3.1.2.1
PROCESS DESCRIPTION: This process sends a message to the user.

PROCESS NAME: Check User Authority
PROCESS NUMBER: 1.2.3.4.1.1
PROCESS DESCRIPTION: This process checks to see if the user is the 'Superuser.'

PROCESS NAME: Validate Run Command
PROCESS NUMBER: 1.2.3.5.1.1
PROCESS DESCRIPTION: This process checks the username and the file for validity.

PROCESS NAME: Validate List Command
PROCESS NUMBER: 1.2.3.5.1.2
PROCESS DESCRIPTION: This process checks the username and the file for validity.

PROCESS NAME: Validate Print Command
PROCESS NUMBER: 1.2.3.5.1.3
PROCESS DESCRIPTION: This process checks the username and the file for validity.

PROCESS NAME: Validate Delete Command
PROCESS NUMBER: 1.2.3.5.1.4
PROCESS DESCRIPTION: This process checks the username and the file for validity.

PROCESS NAME:Get File
PROCESS NUMBER: 1.2.3.5.2.1
PROCESS DESCRIPTION: This process checks the space, retrieves a file from secondary memory, places into main memory, and builds a Process Control Block.

PROCESS NAME: Process Scheduler
PROCESS NUMBER: 1.2.3.5.2.2
PROCESS DESCRIPTION: This process retrieves the unblocked Process Control Blocks in the I/O Wait Queue and places them into the Ready Queue, gets the next process to be executed, and executes the process. If there is no process that is ready to run, then control is returned to the calling module.

PROCESS NAME: Check Filename
PROCESS NUMBER: 1.2.3.5.1.1.1
PROCESS DESCRIPTION: This process determines if a file is located in secondary memory.

PROCESS NAME: Check Username
PROCESS NUMBER: 1.2.3.5.1.1.2
PROCESS DESCRIPTION: This process determines if the user has authority to access the file.

PROCESS NAME: Check Run Filename
PROCESS NUMBER: 1.2.3.5.1.1.3
PROCESS DESCRIPTION: This process determines if a file is an executable file.

PROCESS NAME: Check Space
PROCESS NUMBER: 1.2.3.5.2.1.1
PROCESS DESCRIPTION: This process determines if there exists enough space for an incoming file.

PROCESS NAME: Build PCB
PROCESS NUMBER: 1.2.3.5.2.1.2
PROCESS DESCRIPTION: This process builds a Process Control Block for the command.

PROCESS NAME: Read File
PROCESS NUMBER: 1.2.3.5.2.1.3
PROCESS DESCRIPTION: This process reads a file from secondary memory and places it into main memory.

PROCESS NAME: Process I/O Wait Queue
PROCESS NUMBER: 1.2.3.5.2.2.1
PROCESS DESCRIPTION: This process takes those processes that are finished with their I/O wait out of the I/O Wait Queue and places them into the appropriate Ready Queue.

PROCESS NAME: Get PCB
PROCESS NUMBER: 1.2.3.5.2.2.2
PROCESS DESCRIPTION: This process retrieves the PCB of the next ready process to executed.

PROCESS NAME: Run Process
PROCESS NUMBER: 1.2.3.5.2.2.3
PROCESS DESCRIPTION: This process executes the process of the given PCB.

PROCESS NAME: Open File
PROCESS NUMBER: 1.2.3.5.1.1.1.1
PROCESS DESCRIPTION: This process opens a file located in secondary memory for reading and writing.

PROCESS NAME: Get Username
PROCESS NUMBER: 1.2.3.5.1.1.1.2
PROCESS DESCRIPTION: This process gets the username of the requested file.

PROCESS NAME: Build Error Message
PROCESS NUMBER: 1.2.3.5.1.1.1.3
PROCESS DESCRIPTION: This process constructs an error message that is the transmitted to the user.

PROCESS NAME: Sort Memory Locations
PROCESS NUMBER: 1.2.3.5.2.1.1.1
PROCESS DESCRIPTION: This process arranges the memory locations of all jobs in main memory from smallest to largest.

PROCESS NAME: Insert PCB
PROCESS NUMBER: 1.2.3.5.2.1.2.1
PROCESS DESCRIPTION: This process inserts the given PCB into the appropriate queue.

PROCESS NAME: Locate Unblocked Processes
PROCESS NUMBER: 1.2.3.5.2.2.1.1
PROCESS DESCRIPTION: This process locates all PCBs in the I/O Wait Queue that are no longer in an I/O wait state.

PROCESS NAME: Delete PCB From I/O Wait Queue
PROCESS NUMBER: 1.2.3.5.2.2.1.2
PROCESS DESCRIPTION: This process deletes the given PCBs from the I/O Wait Queue.

PROCESS NAME: Check Ready Queue
PROCESS NUMBER: 1.2.3.5.2.2.2.1
PROCESS DESCRIPTION: This process checks the Ready Queues for a ready PCB and returns the Ready Queue's number.

PROCESS NAME: Get PCB From Queue
PROCESS NUMBER: 1.2.3.5.2.2.2.2
PROCESS DESCRIPTION: This process retrieves the PCB from the given Ready Queue.

Appendix D

Data Dictionary for AMOS

This appendix contains the data flow entrys that are passed between the structure chart modules. The structure charts are located in Appendix B.C

1. DATA NAME: AMOS

   This is the object code of the operating system. It is transferred to main memory (at a set location) from secondary memory.

2. DATA NAME: AMOS-Global-Variables

   These are the data items that are used by the operating system and contains all of the data flow items.

3. DATA NAME: Authorized/Unauthorized-User
   ALIASES: Error-Flag

   This is a flag that informs the operating system that the user, who is requesting a system command operation, is or is not the 'Superuser.'

4. DATA NAME: Command-Help

   This is a request for command information.

5. DATA NAME: Command-Information

   This is the command information that was requested by the user

6. DATA NAME: Command-Line
   ALIASES: New-System-Information

   The data sent to the operating system by the user that is terminated by a carriage return. It will contain a command and any necessary parameters.

7. DATA NAME: Command-Table

   All of the parameters from the Command Line (6) and any other parameters that are acquired by any promting routine.

8. DATA NAME: Data-Base-Information

These are the initial values that the AMOS-Global-Variables (2) are set.

9. DATA NAME: Error-Flag

This is a flag that is sent to the Error routine to build an Error-Message (10).

10. DATA NAME: Error-Message

This is a message informing the user that an error has occurred and what it was.

11. DATA NAME: Filename

This is the name of a file that has an operation that is to be performed on it. (such as Run or List)

12. DATA NAME: File-Descriptor

This is an integer indicating where the file is located in a buffer of all open files.

13. DATA NAME: File's-Username

This is the username of the file that is being requested by a user.

14. DATA NAME: I/O-Wait-Queue

This is the pointer to the I/O Wait Queue.

15. DATA NAME: Legal/Illegal-User
ALIASES: Error-Flag

This is a flag indicating that the user has access, or doesn't have access, to AMOS.

16. DATA NAME: Logged-In
ALIASES: Message-Code

This is a flag indicating that the user has been properly logged-in.

17. DATA NAME: Logged-Out
ALIASES: Message-Code

This is a flag indicating that the user has been properly logged-out.

18. DATA NAME: Memory-Location

The location in main memory that a file is located or is being sent.

19. DATA NAME: Message-Code

This is a flag that is used by the Build Message routine to build a message that is sent to the user.

20. DATA NAME: No-Space-Available
ALIASES: Error-Flag

This is a flag indicating that there isn't enough available memory space for the execution of the process.

21. DATA NAME: Ordered-Memory-Locations

This is the table of the available memory partitions ordered by size and is used by the Memory Manager.

22. DATA NAME: Password

This is the user's unique key to the operating system. It can be changed by the user and is entered by the user.

23. DATA NAME: PCB (Process Control Block)

This is a table that is used by the Process Manager to keep track of all the processes that are submitted to run.

24. DATA NAME: Portno

The port number that a message is being sent or a command is being received.

25. DATA NAME: Process-Aborted
ALIASES: Error-Flag

This is a flag that is sent to the user indicating that the submitted process was aborted.

26. DATA NAME: Process-Memory-Locations

This is the table of unordered available memory partitions.

27. DATA NAME: Program-Too-Large-For-Memory
ALIASES: Not-Too-Large/Too-Large,
Error-Flag

This is a flag to indicate to the Memory Manager that the incoming file is not small enough for all of main memory.

28. DATA NAME: Prompt
ALIASES: Message, System-Change

This is a message to the user to indicate that information is to be entered. This information can include the Username (34) or Password (22).

29. DATA NAME: Queues-Empty/Not-Empty

This is a flag indicating that the Ready Queues are empty, or not empty.

30. DATA NAME: Queue-Number

This is a number that indicates to the Process Manager which Ready Queue has the next process to run.

31. DATA NAME: Syntax-Error
ALIASES: Error-Flag

This is a control flag to indicate that a command is not found. This control flag is transmitted to an error handling routine.

32. DATA NAME: System-Help

This is a request for system information.

33. DATA NAME: System-Information

This is the system information that was requested by the user.

34. DATA NAME: Username
ALIASES: Portno's-Username,
Superuser's-Username

This is the user's indentification used to log onto the operating system. It cannot be changed by the user and is entered by the system's 'Superuser.'

35. DATA NAME: Valid/Invalid-Filename
ALIASES: Error-Flag

This is a flag that indicates that the file the user has requested an operation on exists, or doesn't exist.

36. DATA NAME: Valid/Invalid-Username
ALIASES: Error-Flag

This is a flag that indicated that the file the user is requesting is, or isn't, their file.

Appendix E

AMOS Source Code

This appendix contains the source code for AMOS. It is in the file AMOS.C on the VMS O/S's disk storage. The documentation in the code consists of a header for each subroutine and comments throughout the C language code. The header is based on Dr. Gary Lamont's standards that were given in EE 6.86, Information Structures.

```
/*****************************************************************/
/*  Title:   AMOS: AFIT Multiprogramming Operating System        */
/*                                                               */
/*  Date: 31 August 1983                                         */
/*  Version: 1.0                                                 */
/*                                                               */
/*  Filename:  AMOS.C                                            */
/*  Function:  This is a multiprogramming operating system       */
/*             for sixteen-bit microprocessor systems.           */
/*  Calling Subroutines:  When implemented on a micro-           */
/*             processing system,  a boot program will load      */
/*             this O/S and will proceed to execute the O/S      */
/*  Authors:  Paul E. Cruser and Ronald K. Miller                */
/*                                                               */
/*****************************************************************/
/**/
/**/
/*****************************************************************/
/*    The following is the Global Data Base that will be         */
/*    used throughout the operating system.                      */
/*****************************************************************/
/**/

#include stdio.h      /*  standard input-output library    */
#define  superuser    "SUPERMAN"  /* this is the superusers*/
                                  /* username             */
#define  sprpass      "MOONBEAK" /* this is the superusers */
                                  /* password             */
#define  user1        "CRUSERP " /* username for first user*/
#define  user2        "MILLERRK" /*username for second user*/
#define  pass1        "GCS-83D " /* password for first user*/
#define  pass2        "LINDAWM " /*password for second user*/
#define  noports      4   /*  this is the number of online */
                          /*  terminal ports               */
#define  deviceports  0   /*  this is the number of online */
                          /*  device ports                 */
#define  begin        {   /*  personal preference for      */
#define  end          }   /*  easier coding                */
#define  BEGINUSER    13  /*  used in the directory to mark*/
                          /*  the column                   */
#define  ENDUSER      20  /*  of the beginning and ending  */
                          /*  of the username              */
#define  DIRTRACK     0   /*  Directory track number       */
#define  DIRSECTOR    0   /*  Directory sector number      */
#define  ERROR        -1  /*  file can't be opened flag    */
#define  NUMSEC       16  /*  number of sectors on a disk  */
#define  BUFLENGTH    40  /*  number of buffer rows for the*/
                          /*  directory                    */
#define  BUFSIZE      24  /*  number of buffer column f.t.d*/
#define  MESSIZE      40  /*  size of the message          */
#define  BEGSIZE      23  /*  byte location of file size   */
                          /*  found in the directory buffer*/
```

```
#define  BEGTRACK     21  /*  byte location of the first   */
                          /*  track in the directory buffer*/
#define  BEGSECTOR    22  /*  byte location of first sector*/
                          /*  in the directory buffer      */
#define  OFFSET       40  /*  ASCII offset                 */
#define  MAXJOBS      4   /*  Maximum no. of jobs allowed  */
                          /*  on the system                */
#define  NAMESIZE     12  /*  The size of a filename       */
#define  DISKSTAT     0   /*  The disk status bit          */
#define  DISKRDA      0   /*  The disk ready bit           */
#define  DISKPORT     0   /*  The disk dataport            */
#define  COMMSIZE     5   /*  The max size of a command    */
#define  PARASIZE     12  /*  The max size of a parameter  */
#define  BASE_ADDRESS 200 /* Start address of user memory */
#define  TOP_ADDRESS  0xFFFF /* End address of main memory */
#define  BYTE_SIZE    128  /* Number of bytes in a block   */
                           /* from the disk                */


/* the following structure defines the process control     */
/* blocks temppcb and pcb[] are pcb that will be used by   */
/* the scheduler                                           */

struct  z8pcb  {
                struct z8pcb *next_cb,
                                /* next pcb in the queue */
                             *previous_cb;
                             /* previous pcb in the queue */

                int    priority,
                  /* used to determine queue to put into */
                                /*   0: system queue     */
                                /*   1: readyl queue     */
                                /*   etc.                */

                   current_q,
                  /* indicates what queue it resides in */
                                /*  0: system queue      */
                                /*  1: readyl queue      */
                                /*  etc.                 */
                                /* -1: i/o wait queue    */

                   process_data,
                  /* to be used in later implementation */
                  /* for the address of data workspace  */
                  /* for the process                    */

                   offset_address,
                    /* where the beginning address of */
                     /* allocated memory is located   */

                   final_address,
```

```
                        /* where the final address of the */
                        /* allocated memory is located    */

                    command_type,
                  /* this is set to tell the system what */
                   /* kind of command is being executed  */
                        /*  -1: SYS command               */
                        /*   0: RUN command               */
                        /*   1: LIST command              */
                        /*   2: PRINT command             */
                        /*   3: DEL  command              */
                        /*   4: DIR  command              */
                        /*   5: EDIT command              */
                        /*  note: EDIT isn't available    */
                        /*  and others may be added as    */
                        /*  they are needed               */

                    port_of_origin,
                   /* port number that the pcb's process */
                   /* originated from                    */

                    io_status;
                  /* indicates if process is in an io wait */
                  /* or not, and determines if the process */
                  /* should be taken out of the io wait q  */
                  /*     0: not waiting for io             */
                  /*     1: waiting for io                 */
                }
               pcb[noports];

/*  the following is the headers for the I/O Wait queue */
/*  and the Ready queue.  This is where other queues    */
/*  would be defined when they are added later.         */

struct qheader {
                struct z8pcb  *start,
                        /* pointer to first pcb in list */
                           *ending;
                        /* pointer to last pcb in list  */

                int    qcount;
               }
               iowaitq,   /* header for I/O Wait queue     */
               systemq,   /* header for system ready queue */
               readylq;   /* header for readyl queue       */


/*   the following structure defines the ports' data table */

struct  portdata {
                 int  statport,
                       /* this is the status port address */
```

```
                    d_aport,
                       /* this is the data port address */
                    sendbit,
                          /* this is the send bit mask */
                    recvbit;
                       /* this is the receive bit mask */
                }
               ports[noports+1];    /* ports' data table */
                     /* the noports+1 ports will be for */
                     /* the printer port information    */

/*   the following structure defines the device table      */

struct  dev_table {
                     char  device_type[10];
                                  /* used in the SYS DEV  */
                                  /* to indicate what the */
                                  /* device is            */

                     int   controlport,
                          /* the address of the control port */
                           port_data;
                             /* the address of the data port */
                     }
                     device_table[deviceports];

/* the following structure defines the terminal-user table */

struct  userblock {
                     int loggedon,  /* the logged on flag:   */
                         jobrunning;  /* is a job running:   */
                                    /*           0-no,1-yes */

                     char usernm[8];   /* the logged on user */
                    }
                    userblocks[noports];

/*  the userblocks' subscript (noports) will be used to    */
/*  indicate which terminal is being used by the usernm[]  */

/*   the following structure defines the usertable    */
/*   usertable[] is the table                         */

struct  usrtable {
                     char  username[8],
                           password[8];
                    }
                    usertable[40];

/*  the following structure defines the de: .e table  */
/*  this table is used to delete files that users     */
/*  want deleted that are on the same sector/track    */
```

```
/*  this will save time writing to the disk and will  */
/*  also prevent overwrite                            */

struct  del_table {
                    int  track, sector, /* track and sector */
                         portno[MAXJOBS], /* port numbers of*/
                                 /* jobs using delete for  */
                                 /* this sector and track  */
                         del_count,
                            /* number of deletes to perform */
                             /* for this track and sector   */
                         row[MAXJOBS];
                              /* array of the rows to be    */
                             /* deleted on the given track */
                              /* and sector                 */
                   }
                   delete_table[MAXJOBS];


/*  the following are various integer and character        */
/*  declarations used throughout the system                */

       int    poll_portno,
              /* current port which polling routine starts */
              portno,
             /* current port number that is being accessed */
              no_users_on_sys,
                /* number of users currently on the system */
              no_of_users,
             /* number of users that can access the system */

              cmd,       /* code for user command type     */
                         /*   1: RUN                       */
                         /*   2: LIST                      */
                         /*   3: PRINT                     */
                         /*   4: DEL (Delete)              */
                         /*   5: DIR (Directory)           */

              memory_loc, /* The memory location for a file*/
              fd, /* file descriptor used  on opened files */
              finished, /* boolean indicating entire file  */
                        /* has been read                   */
              size,    /* number of blocks used by a file  */
              dtrack,  /* directory track to read          */
              dsector, /* directory sector to read         */
              del_track, /* dir track of file for deletion */
              del_sector, /* dir sector of file for delet. */
              number_jobs,/* number of jobs in main memory */
              begin_address[MAXJOBS+1], /* beginning addr. */
                   /* of each jobs main memory allocation */
              end_address[MAXJOBS], /* end addr. of each   */
                           /* jobs main memory allocation */
```

```
                order[MAXJOBS+1]; /* indices of the sorted   */
                                  /* beginning addr.  location*/

        char    command_line[32], /* command line entered by */
                                  /* the user */
                file_username[8],/*username of requested file*/
                opened_files[4] [32], /* info of opened files*/
                name[12],    /* filename from the directory  */
                file[12],    /* filename sent from the user  */
                message[MESSIZE]; /* message tobe transmitted*/


/*  the following is the structure definition for the      */
/*  command table which will be used through the validation*/
/*  of the command and setting up of the pcbs              */

struct comm_table {
                    char command[8],
                         parameter1[12],
                         parameter2[12];

                    int  numparam;
                             /*  0 = only a command        */
                             /*  1 = one param plus command */
                             /*  2 = two param plus command */
            /*  int  terminal;  this may be used, but portno */
              /*  should be used with no forseeable problem */
                    }
                    command_table;
```

```
/*****************************************************************/
/*                                                               */
/*                          MAIN                                 */
/*                                                               */
/*     Date: 31 August 1983                                      */
/*     Version: 1.0                                              */
/*                                                               */
/*     Name:  main                                               */
/*     Module Number: 1.2                                        */
/*     Function: This is the module that initializes the data    */
/*               base, monitors the consoles and validates the   */
/*               commands.                                       */
/*                                                               */
/*     Calling Modules: None                                     */
/*     Modules Called: initialize_data_base, p_comm_line,        */
/*                     and det_valid_comm                        */
/*                                                               */
/*     Global Variables Used:  None                              */
/*     Global Variables Changed: None                            */
/*                                                               */
/*     Author: Paul E. Cruser and Ronald K. Miller               */
/*     System: VAX 11/780, VMS 0/S and UNIX: for testing, only   */
/*                                                               */
/*****************************************************************/

main()
begin
   initialize_data_base();
   for (;;)
     begin
       p_comm_line();
       det_valid_comm();
     end
end
```

```
/*****************************************************************/
/*                                                               */
/*                    INITIALIZE DATA BASE                       */
/*                                                               */
/*     Date:  1 September 1983                                   */
/*     Version: 1.0                                              */
/*                                                               */
/*     Name: initialize_data_base                                */
/*     Module Number:  1.2.1                                     */
/*     Function:   To enter those initial parameters, that are   */
/*               necessary for the operation of the o/s, into    */
/*               the data base.                                  */
/*                                                               */
/*     Calling Modules:  main                                    */
/*     Modules Called:   none                                    */
/*                                                               */
/*     Global Variables Used:  temppcb, pcb[], ports[],          */
/*                             userblocks[], usertable[],        */
/*                             no_users_on_sys, and no_of_users  */
/*     Global Variables Changed: all of the ones used            */
/*                                                               */
/*     Author:  Paul E. Cruser                                   */
/*     System:  VAX 11/780, VMS O/S and UNIX O/S: testing only   */
/*                                                               */
/*****************************************************************/

initialize_data_base()
begin
int  count;
/* the following are initialization of the status & data ports' */
/* addresses and the masks for the send and receive bits        */
/* ports 0-3 are console ports; port 4 is a printer port        */
   ports[0].statport = 0;
   ports[0].dataport = 0;
   ports[0].sendbit = 0;
   ports[0].recvbit = 0;
   ports[1].statport = 0;
   ports[1].dataport = 0;
   ports[1].sendbit = 0;
   ports[1].recvbit = 0;
   ports[2].statport = 0;
   ports[2].dataport = 0;
   ports[2].sendbit = 0;
   ports[2].recvbit = 0;
   ports[3].statport = 0;
   ports[3].dataport = 0;
   ports[3].sendbit = 0;
   ports[3].recvbit = 0;
   ports[4].statport = 0;
   ports[4].dataport = 0;
   ports[4].sendbit = 0;
   ports[4].recvbit = 0;
```

```
/*  when more ports are made available, then they are to */
/*  be added on to this initialization list   */

/*  the following is the initialization of the status bits of   */
/*  the userblocks (the structures that tell the system who is  */
/*  logged onto which terminal)                                 */
/*  the loggedon and jobrunning will both be initialized to 0   */

 count = 0;
 while (count < noports)
  begin
    userblocks[count].loggedon = 0;
    userblocks[count].jobrunning = 0;
    count = count + 1;
  end

/*  the following is the initialization of the usernames    */
/*  and the passwords                                       */

 strcpy(usertable[0].username,superuser);
 strcpy(usertable[0].password,sprpass);
 strcpy(usertable[1].username,user1);
 strcpy(usertable[1].password,pass1);
 strcpy(usertable[2].username,user2);
 strcpy(usertable[2].password,pass2);
   /* etc. */

/* the following is the initialization of the queue counter for */
/* the I/O wait, System, and Ready1 queues                      */

 iowaitq.qcount = 0;
 systemq.qcount = 0;
 readylq.qcount = 0;

/* the following initializations are for:                       */
/* number of users, which is currently 3                        */
/* number of users on the system, which is 0                    */
/* number of jobs on the system, which is 0                     */
/* portno, set to the first port - 0                            */
/* the portno will be changed in the polling routine            */
/* the number of users on the system will change as the users   */
/* log on and off of the system                                 */

 no_of_users = 3;
 no_users_on_sys = 0;
 number_jobs = 0;
 portno = 0;
 poll_portno = 0;

/*  the following is the initialization of the begin address */
/*  and end address of the available memory space. These are */
/*  used to allocate and deallocate memory (Memory Mgt.)     */
```

```
    begin_address[MAXJOBS] = TOP_ADDRESS;
    end_address[MAXJOBS] = TOP_ADDRESS;

    return(1);

end  /*  the end of the initialization subroutine  */
```

```
/*****************************************************************/
/*                                                               */
/*                        LOGIN-USER                             */
/*                                                               */
/*     Date: 13 September 1983                                   */
/*     Version: 1.1                                              */
/*                                                               */
/*     Name: login-user                                          */
/*     Module Number: 1.2.3.1                                    */
/*     Function: This module will determine if the user is       */
/*              - to log onto the system and then enters the     */
/*              his username into the userblock table.           */
/*                                                               */
/*     Calling Modules: det_valid_comm                           */
/*     Modules Called:  build_message, get_command_line, error,  */
/*                      checkuser, and  strcpy                   */
/*                                                               */
/*     Global Variables Used:  userblocks[].usernm, portno,      */
/*                             userblocks[].loggedon,            */
/*                             no_users_on_sys,                  */
/*                             and command_line                  */
/*     Global Variables Changed:  userblocks[].usernm and        */
/*                                no_users_on_sys                */
/*                                                               */
/*     Author:  Paul E. Cruser                                   */
/*     System:  VAX 11/780, VMS O/S and UNIX O/S: testing, only  */
/*                                                               */
/*****************************************************************/

login_user()
begin
#define  unmessage  6
#define  pwmessage  7
#define  illegal_user   4
#define  login_complete 9
int i,j;
char un[8],
     pw[8];

  if (userblocks[portno].loggedon == 0)
   begin
     build_message(unmessage);   /* username prompt */
     get_command_line();  /* get the username from the user */
     i = 0;
     while (i < 8)     /* take the username from the comm line */
       if (command_line[i] != '\n')
        un[i] = command_line[i];
       else
         begin     /* fill the rest of un[] with blanks */
          for (j=i;7;j++)
            un[i] = ' ';
          i = 8;
```

```
        end

      build_message(pwmessage);   /* password prompt */
      get_command_line();  /* get the password from the user */
      i = 0;
      while (i < 8)    /* take the password from the comm line */
        if (command_line[i] != '\n')
         un[i] = command_line[i];
        else
          begin     /* fill the rest of pw[] with blanks */
           for (j=i;7;j++)
             un[i] = ' ';
           i = 8;
          end

      if (checkuser(un,pw))
                    /* check to see if user can get on system */
       begin
       /* copy the necessary data into the next available      */
       /* user block                                           */
         strcpy(userblocks[no_users_on_sys].usernm,un);
         userblocks[no_users_on_sys].loggedon = 1;
         no_users_on_sys += 1;
         build_message(login_complete);
         return(1);   /* one returned if login successful */
       end
      else
       begin
       /* send illegal user message to the console trying      */
       /* to log in                                            */
         error(illegal_user);
         return(0);   /* zero returned if login unsuccessful     */
                      /* ie. wrong password or invalid user name */
       end
    end
   else
    return(2);        /* two returned if login not necessary    */
                      /* ie. the user is already logged on      */
end
```

```
/*****************************************************************/
/*                                                               */
/*                        LOGOUT-USER                            */
/*                                                               */
/*     Date:   1 September 1983                                  */
/*     Version: 1.0                                              */
/*                                                               */
/*     Name:  logout_user                                        */
/*     Module Number:  1.2.3.2                                   */
/*     Function:  To clear the userblock of the username and     */
/*              to set the loggedon and jobrunning flags to      */
/*              'no'.  The jobrunning will be set as a pre-      */
/*              caution.                                         */
/*                                                               */
/*     Calling Modules:  det_valid_comm                          */
/*     Modules Called: build_message, strcpy                     */
/*                                                               */
/*     Global Variables Used: userblocks, portno, and            */
/*                            no_users_on_sys                    */
/*     Global Variables Changed:  userblocks and                 */
/*                                no_users_on_sys                */
/*                                                               */
/*     Author:  Paul E. Cruser                                   */
/*     System:  VAX 11/780, VMS O/S and UNIX O/S: testing only   */
/*                                                               */
/*****************************************************************/

logout_user()
begin
#define  logoutmessage  8
int i;
   for (i=0;7;i++)
       userblocks[portno].usernm[i] = ' ';
   userblocks[portno].loggedon = 0;
   userblocks[portno].jobrunning = 0;
   no_users_on_sys = no_users_on_sys - 1;
   build_message(logoutmessage);
   return(1);
end
```

```
/*****************************************************************/
/*                                                               */
/*                    PARSE COMMAND LINE                         */
/*                                                               */
/*     Date: 13 September 1983                                   */
/*     Version: 1.1                                              */
/*                                                               */
/*     Name:  p_comm_line                                        */
/*     Module Number:  1.2.2                                     */
/*     Function: To retrieve the data line that is entered by    */
/*               at a terminal.  The lower case letters will     */
/*               then be changed to upper case, only for the     */
/*               command and not the parameters                  */
/*                                                               */
/*     Calling Modules:  main                                    */
/*     Modules Called:   get_command_line, build_parse_table,    */
/*                       process_scheduler, poll                 */
/*                                                               */
/*     Global Variables Used:  none                              */
/*     Global Variables Used:  none                              */
/*                                                               */
/*     Author: Paul E. Cruser                                    */
/*     System: VAX 11/780, VMS O/S and UNIX O/S: testing only    */
/*                                                               */
/*****************************************************************/

p_comm_line()
begin
#define  process_abort    9
#define  processing_done  10
   if (poll())
    begin
      get_command_line();   /* get comm line from the user  */
      build_parse_table();  /* have the comm line parsed and */
                            /* saved                         */
    end
   else
    if (!process_scheduler())
       error(process_abort);
    else
       build_message(processing_done);
end
```

```
/*****************************************************************/
/*                                                               */
/*                          POLL                                 */
/*                                                               */
/*     Date:   11 October 1983                                   */
/*     Version:  1.1                                             */
/*                                                               */
/*     Name:  poll                                               */
/*     Module Number: 1.2.2.1                                    */
/*     Function:  To poll the ports that the users consoles      */
/*                will be communicating through.  It will        */
/*                return a 1 if a port is sending something      */
/*                or a 0 if it has checked each port once and    */
/*                has not gotten a response from any of them     */
/*                                                               */
/*     Calling Modules:  p_comm_line                             */
/*     Modules Called:   none                                    */
/*                                                               */
/*     Global Variables Used: portno, ports[]                    */
/*     Global Variables Changed:  portno                         */
/*                                                               */
/*     Author:  Paul E. Cruser                                   */
/*     System:  VAX 11/780, VMS O/S and UNIX O/S: testing only   */
/*                                                               */
/*****************************************************************/

poll()
begin
int pno,  /* temporary port number variable */
    yes,  /* flag to tell if a port needs to be tended to */
    counter,  /* count how many ports have been checked */
    tempstat, /* temp. storage for the status ports' contents */
    rstat;    /* status byte masked */
   yes = 0;
   counter = 0;
   pno = poll_portno;
   while (!yes)
    while ((counter < noports) && (!yes))
     begin
       if (userblocks[pno].jobrunning != 1)
        begin
          rstat = (inp(ports[pno].statport) & ports[pno].recvbit);
          if (rstat = 0)
           begin
             yes = 1;
             portno = pno;
             return(1);
           end
          else if(pno == noports-1)
                  pno = 0;
               else
                  pno = pno + 1;
```

```
        end
      counter += 1;
    end
  return(0);
end
```

```
/*****************************************************************/
/*                                                               */
/*                      SYSTEM_CHANGE                            */
/*                                                               */
/*     Date:  3 September 1983                                   */
/*     Version: 1.0                                              */
/*                                                               */
/*     Name:  system_change                                      */
/*     Module Number: 1.2.3.4                                    */
/*     Function:  This will verify that the user is authorized   */
/*                to make the system changes that are requested  */
/*                and then makes those changes using a menu      */
/*                system, if a menu is needed.                   */
/*                                                               */
/*     Calling Modules: det_valid_comm                           */
/*     Modules Called:  build_pcb, error, adduser, and deluser   */
/*                                                               */
/*     Global Variables Used: superuser, userblocks[].username,  */
/*                            portno                             */
/*     Global Variables Changed: none                            */
/*                                                               */
/*     Author:  Paul E. Cruser                                   */
/*     System:  VAX 11/780, VMS O/S and UNIX O/S: testing only   */
/*                                                               */
/*****************************************************************/

system_change()
begin
#define not_superuser 5
char  *systemcomm;
   if (stringcmp(superuser,userblocks[portno].usernm))
    begin
      build_pcb(-1);
      systemcomm = "ADDUSER";
      if (stringcmp(systemcomm,command_table.parameter1))
         adduser();
      else
       begin
         systemcomm = "DELUSER";
         if (stringcmp(systemcomm,command_table.parameter1))
            deluser();
      else
       begin
         /* to be updated when other commands */
         /* are necessary for the system      */
       end /* last else */
       end /* previous else */
    end /* first if */
   else
    error(not_superuser);
end
```

```
/*****************************************************************/
/*                                                               */
/*                        HELP-USER                              */
/*                                                               */
/*     Date: 3 September 1983                                    */
/*     Version: 1.0                                              */
/*                                                               */
/*     Name:  help_user                                          */
/*     Module Number: 1.2.3.3                                    */
/*     Function:   This checks to see if the help can be pro-    */
/*                 vided and gives out the information or a      */
/*                 message is sent to the user that no info is   */
/*                 available.                                    */
/*                                                               */
/*     Calling Modules: det_valid_comm                           */
/*     Modules Called:  build_message, stringcmp, users_on_line  */
/*                      and devices_available                    */
/*                                                               */
/*     Global Variables Used:   portno, usertable                */
/*     Global Variables Changed: none                            */
/*                                                               */
/*     Author:  Paul E. Cruser                                   */
/*     System:  VAX 11/780, VMS O/S and UNIX O/S: testing only   */
/*                                                               */
/*****************************************************************/

help_user()
begin
#define no_help           0
#define run_format        1
#define list_format       2
#define print_format      3
#define delete_format     4
#define directory_format  5
int  help_set;
char *users,*devices,*run,*list,*print,*delete,*directory;
   users = "USERS"; /* request to list the users on the system  */
   devices = "DEV"; /* request to list the devices on-line      */
   run = "RUN";     /* req. to show the format for run comm.    */
   list = "LIST";   /* req. to show the format for list comm.   */
   print = "PRINT"; /* req. to show the format for print comm.  */
   delete = "DEL";  /* req. to show the format for delete comm. */
   directory = "DIR"; /* req. to show the format for dir. comm. */
   help_set = 0;
   if (stringcmp(users,command_table.parameter1))
      help_set = 1;  /* system inquiry */
   else
    if (stringcmp(devices,command_table.parameter1))
      help_set = 2;  /* system inquiry */
   else
    if (stringcmp(run,command_table.parameter1))
      help_set = 3;  /* command inquiry */
```

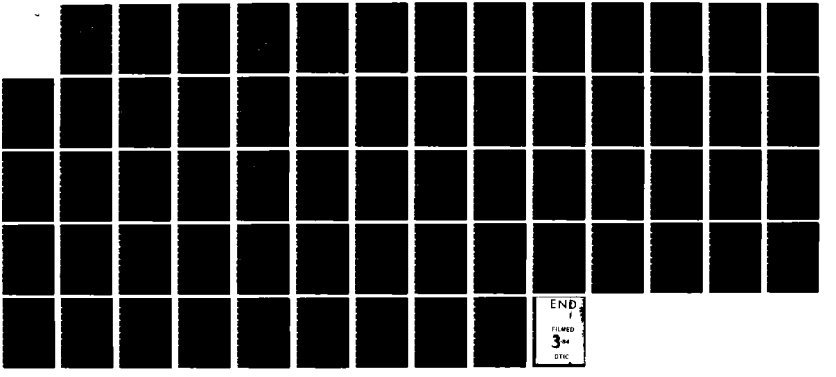
```
        else
         if (stringcmp(list,command_table.parameter1))
           help_set = 4;  /* command inquiry */
        else
         if (stringcmp(print,command_table.parameter1))
           help_set = 5;  /* command inquiry */
        else
         if (stringcmp(delete,command_table.parameter1))
           help_set = 6;  /* command inquiry */
        else
         if (stringcmp(directory,command_table.parameter1))
           help_set = 7;  /* command inquiry */
        switch(help_set)
           begin
             case 0: build_message(no_help);
             case 1: users_on_line();
             case 2: devices_available();
             case 3: build_message(run_format);
             case 4: build_message(list_format);
             case 5: build_message(print_format);
             case 6: build_message(delete_format);
             case 7: build_message(directory_format);
             default: build_message(no_help);
           end
    end
```

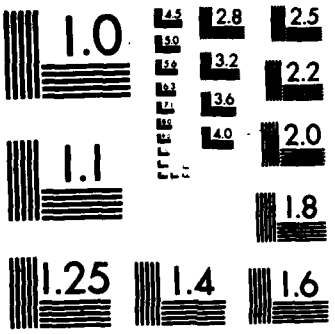
```
/*********************************************************************/
/*                                                                   */
/*                            ADDUSER                                */
/*                                                                   */
/*     Date:  7 September 1983                                       */
/*     Version: 1.0                                                  */
/*                                                                   */
/*     Name:  adduser                                                */
/*     Module Number: 1.2.3.4.2A                                     */
/*     Function:  Adds a new user's username and password            */
/*                                                                   */
/*     Calling Modules:  system_change                               */
/*     Modules Called:  none                                         */
/*                                                                   */
/*     Global Variables Used: no_of_users, usertable                 */
/*     Global Variables Changed: no_of_users, usertable              */
/*                                                                   */
/*     Author:  Paul E. Cruser                                       */
/*     System:  VAX 11/780, VMS O/S and UNIX O/S: testing only       */
/*                                                                   */
/*********************************************************************/

adduser()
begin
#define blank ' '
int     cnt;
   no_of_users += 1.
   cnt = 0;
   while (cnt <= 7)
    begin
       cnt += 1;
       /* read in the username into */
       /* usertable[no_of_users-1].username */
    end
   cnt = 0;
   while (cnt <= 7)
     begin
       usertable[no_of_users-1].password[cnt] = blank;
       cnt += 1;
     end
end
```

```
/*****************************************************************/
/*                                                               */
/*                          DELUSER                              */
/*                                                               */
/*     Date: 7 September 1983                                    */
/*     Version: 1.0                                              */
/*                                                               */
/*     Name:   deluser                                           */
/*     Module Number: 1.2.3.4.2B                                 */
/*     Function:  Delete a user's username and password from     */
/*                the usertable                                  */
/*                                                               */
/*     Calling Modules:  system_change                           */
/*     Modules Called:   none                                    */
/*                                                               */
/*     Global Variables Used:   no_of_users, usertable           */
/*     Global Variables Changed:  no_of_users, usertable         */
/*                                                               */
/*     Author:  Paul E. Cruser                                   */
/*     System:  VAX 11/780, VMS O/S and UNIX O/S: testing only   */
/*                                                               */
/*****************************************************************/

deluser()
begin
  int cont,i,j;
  char deletename[8];
  /* read deletename from superuser */
   for (cont = 1;no_of_users-1;cont++)
      if (stringcmp(usertable[cont].username,deletename))
         i = cont;
   for (cont = i;no_of_users-2;cont++)
 /* shift the table down over the user's id */
      begin
        for (j = 0;7;j++)
          begin
            usertable[cont].username[j]=
                              usertable[cont+1].username[j];
            usertable[cont].password[j]=
                              usertable[cont+1].password[j];
          end
      end
   no_of_users -= 1;
end
```

AD-A138 083 DETAILED DESIGN AND IMPLEMENTATION OF A MULTIPROGRAMMING OPERATING SYSTEM..(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. 3/3

UNCLASSIFIED P E CRUSER DEC 83 AFIT/GCS/EE/83D-5 F/G 9/2 NL





```
/*****************************************************************/
/*                                                               */
/*                        USERS_ON_LINE                          */
/*                                                               */
/*     Date:  9 September 1983                                   */
/*     Version: 1.0                                              */
/*                                                               */
/*     Name:  users_on_line                                      */
/*     Module Number: 1.2.3.3.1A                                 */
/*     Function:    Lists the users that are logged into the     */
/*                  system and the terminals they are using      */
/*                                                               */
/*     Calling Modules:  help_user                               */
/*     Modules Called:   transmit_message                        */
/*                                                               */
/*     Global Variables Used: noports, userblocks, MESSIZE       */
/*     Global Variables Changed:  none                           */
/*                                                               */
/*     Author:  Paul E. Cruser                                   */
/*     System:  VAX 11/780, VMS O/S and UNIX O/S: testing only   */
/*                                                               */
/*****************************************************************/

users_on_line()
begin
int duke,i;
char terminal[MESSIZE - 10];
   strcpy(terminal,"                          TERMINAL ");
   for (duke = 0;noports - 1;duke++)
     if (userblocks[duke].loggedon)
       begin
         for (i = 0;7;i++)
           message[i] = userblocks[duke].usernm[i];
         for (i = 0;MESSIZE - 10;i++)
           message[i+7] = terminal[i];
         switch(duke)
             begin
               case 0: { message[MESSIZE-2] = "0"; break; }
               case 1: { message[MESSIZE-2] = "1"; break; }
               case 2: { message[MESSIZE-2] = "2"; break; }
               case 3: { message[MESSIZE-2] = "3"; break; }
             end
         message[MESSIZE-1] = "\n";
         transmit_message(message);
       end
end
```

```
/*****************************************************************/
/*                                                               */
/*                      DEVICES_AVAILABLE                        */
/*                                                               */
/*     Date: 9 September 1983                                    */
/*     Version:  1.0                                             */
/*                                                               */
/*     Name:   devices_available                                 */
/*     Module Number: 1.2.3.3.1B                                 */
/*     Function:    To list the devices that are online          */
/*                                                               */
/*     Calling Modules:  help_user                               */
/*     Modules Called:   transmit_message                        */
/*                                                               */
/*     Global Variables Used:  device_table, MESSIZE             */
/*     Global Variables Changed: none                            */
/*                                                               */
/*     Author:  Paul E. Cruser                                   */
/*     System:  VAX 11/780, VMS O/S and UNIX O/S: testing only   */
/*                                                               */
/*****************************************************************/

devices_available()
begin

/* will be written later */
/* it will be the same as */
/* users_on_line in structure */

end
```

```
/*****************************************************************/
/*                                                               */
/*                      BUILD_MESSAGE                            */
/*                                                               */
/*     Date:  9 September 1983                                   */
/*     Version: 1.0                                              */
/*                                                               */
/*     Name:  build_message                                      */
/*     Module Number:  1.2.3.1.2                                 */
/*     Function:   To build a message to be sent to the user     */
/*                 by using a integer code sent in to indicate   */
/*                 what message is needed.                       */
/*                                                               */
/*     Calling Modules:  login_user, logout_user, help_user      */
/*     Modules Called:   transmit_message                        */
/*                                                               */
/*     Global Variables Used:  message, MESSIZE                  */
/*     Global Variables Changed:  message                        */
/*                                                               */
/*     Author:  Paul E. Cruser                                   */
/*     System:  VAX 11/780, VMS O/S and UNIX O/S: testing only   */
/*                                                               */
/*****************************************************************/

build_message(coded)
int  coded;
begin
char code0[MESSIZE],code1[MESSIZE],code2[MESSIZE],code3[MESSIZE],
     code4[MESSIZE],code5[MESSIZE],code6[MESSIZE],code7[MESSIZE],
     code8[MESSIZE],code9[MESSIZE],code10[MESSIZE];
     strcpy(code0,"No help is available for that command  \n");
     strcpy(code1,"Format: RUN FILENAME (executable file) \n");
     strcpy(code2,"Format: LIST FILENAME                  \n");
     strcpy(code3,"Format: PRINT FILENAME (nonexecutable) \n");
     strcpy(code4,"Format: DEL FILENAME                   \n");
     strcpy(code5,"Format: DIR                            \n");
     strcpy(code6,"USERNAME:                              \n");
     strcpy(code7,"PASSWORD:                              \n");
     strcpy(code8,"Logged out...                          \n");
     strcpy(code9,"Log-in complete...                     \n");
     strcpy(code10,"Processing of job complete...         \n");
     switch(coded)
      begin
        case 0: { strcpy(message,code0); break; }
        case 1: { strcpy(message,code1); break; }
        case 2: { strcpy(message,code2); break; }
        case 3: { strcpy(message,code3); break; }
        case 4: { strcpy(message,code4); break; }
        case 5: { strcpy(message,code5); break; }
        case 6: { strcpy(message,code6); break; }
        case 7: { strcpy(message,code7); break; }
        case 8: { strcpy(message,code8); break; }
```

```
            case 9: { strcpy(message,code9); break; }
            case 10: { strcpy(message,code10); break; }
            default: { error(6); return(0); }
          end
        transmit_message(message);
        return(1);
    end
```

```
/*****************************************************************/
/*                                                               */
/*                        BUILD_PCB                              */
/*                                                               */
/*     Date: 13 September 1983                                   */
/*     Version:  1.0                                             */
/*                                                               */
/*     Name:  build_pcb                                          */
/*     Module Number:  1.2.3.5.2.1.2                             */
/*     Function:   To initialize a pcb for a job/process to      */
/*                 be run or put on a queue                      */
/*                                                               */
/*     Calling Modules: get_file, system_change                  */
/*     Modules Called:  insert_pcb                               */
/*                                                               */
/*     Global Variables Used:pcb[], userblock[], and portno      */
/*     Global Variables Changed: pcb[]                           */
/*                                                               */
/*     Author:  Paul E. Cruser                                   */
/*     System:  VAX 11/780, VMS O/S and UNIX O/S: testing only   */
/*                                                               */
/*****************************************************************/

build_pcb(jobcode)
int jobcode;

/* note: if there is another ready queue added, then another  */
/* parameter would be passed in to give the priority          */

begin
  pcb[portno].port_of_origin = portno;
  pcb[portno].io_status = 1;
  if (jobcode == -1)
   begin   /* if it is a SYS command */
    pcb[portno].priority = 0;  /* this is where the second */
                               /* parameter would be used  */
    pcb[portno].current_q = 0; /* but not here, since it   */
                               /* will always be 0 for SYS */
    pcb[portno].io_status = 0; /* only for SYS:  it will   */
                               /* not have to wait for i/o */
                               /* until time to write to   */
                               /* data base on disk- which */
                               /* is not being coded in    */
                               /* this version of AMOS     */
    pcb[portno].command_type = jobcode;
   end
  else    /* else it is something else */
   begin
    pcb[portno].priority = 1;  /* this is where the second */
                               /* parameter would be used, */
    pcb[portno].current_q = 1; /* as well as here          */
    pcb[portno].command_type = jobcode;
```

```
    end
  userblocks[portno].jobrunning = 1; /* set jobrunning to 1, */
                                     /* so that port will not */
                                     /* be checked in the poll */
                                     /* routine                */

  insert_pcb();
end
```

```
/*****************************************************************/
/*                                                               */
/*                    GET_COMMAND_LINE                           */
/*                                                               */
/*     Date: 13 September 1983                                   */
/*     Version: 1.0                                              */
/*                                                               */
/*     Name:  get_command_line                                   */
/*     Module Number: 1.2.2.2                                    */
/*     Function:   To read in a line from the user's port        */
/*                                                               */
/*     Calling Modules: p_command_line, login_user               */
/*     Modules Called:  getchar(), inp()                         */
/*                                                               */
/*     Global Variables Used: command_line, ports[], portno      */
/*     Global Variables Changed: command_line                    */
/*                                                               */
/*     Author:  Paul E. Cruser                                   */
/*     System:  VAX 11/780, VMS O/S and UNIX O/S: testing only   */
/*                                                               */
/*****************************************************************/

get_command_line()
begin
int q;
      q = 0;
      while ((q<31) &&((command_line[q] = getchar()) != '\n'))
       begin
        q = q + 1;   /* read in the command line */
        while ((inp(ports[portno].statport) &
                              ports[portno].recvbit) == 0);
       end

    /* note: getchar will not be the library routine   */
    /*       it will have to worry about what port to  */
    /*       receive the response. The getchar routine */
    /*       will be replaced by inp(portaddress),     */
    /*       where portaddress is taken from the ports */
    /*       table.                                    */

      if (q = 31) command_line[31] = '\n';
                  /* make sure there is a carriage return     */
 return(1);
end
```

```
/*****************************************************************/
/*                                                               */
/*                        CHECKUSER                              */
/*                                                               */
/*      Date: 13 September 1983                                  */
/*      Version: 1.0                                             */
/*                                                               */
/*      Name:  checkuser                                         */
/*      Module Number: 1.2.3.1.1.3                               */
/*      Function:   To see if the user is logged onto the system */
/*                  The value 1 is returned if not else 0        */
/*                                                               */
/*      Calling Modules: login_user                              */
/*      Modules Called:  stringcmp                               */
/*                                                               */
/*      Global Variables Used: usertable[], noports              */
/*      Global Variables Changed: none                           */
/*                                                               */
/*      Author:  Paul E. Cruser                                  */
/*      System:  VAX 11/780, VMS O/S and UNIX O/S: testing only  */
/*                                                               */
/*****************************************************************/

checkuser(unm,pwd)
char  unm[8],pwd[8];
begin
 int  no,counters;
 no = 1;  /* flag to indicate if username was found */
 counters = 0;  /* step counter for the usertable  */
   while ((no) && (counters < noports))
     begin
      if (stringcmp(usertable[counters].username,unm))
         no = 0;
      if (no) counters += 1;
     end
   if ((!no) && (counters < noports))
      if (stringcmp(usertable[counters].password,pwd))
         return(1);
 return(0);
end
```

```
/*****************************************************************/
/*                                                               */
/*                        INSERT_PCB                             */
/*                                                               */
/*     Date:  15 September 1983                                  */
/*     Version:  1.1                                             */
/*                                                               */
/*     Name:   insert_pcb                                        */
/*     Module Number: 1.2.3.5.2.1.2.1                            */
/*     Function:   To insert a process control block into a      */
/*                 queue.  The queue is determined by the        */
/*                 priority that was set when the pcb was        */
/*                 initialized.                                  */
/*                                                               */
/*     Calling Modules:  build_pcb                               */
/*     Modules Called:   none                                    */
/*                                                               */
/*     Global Variables Used:  pcb[], systemq, readyq, iowaitq   */
/*     Global Variables Changed: systemq, readyq, iowaitq        */
/*                                                               */
/*     Author:  Paul E. Cruser                                   */
/*     System:  VAX 11/780, VMS O/S and UNIX O/S: testing only   */
/*                                                               */
/*****************************************************************/

insert_pcb()
begin
 if (!pcb[portno].io_status)
   switch (pcb[portno].priority)
    begin
      case 0:

/*  insert into the systemq  */

       begin
         if (systemq.qcount == 0)
          begin
            /* systemq.start = pcb[portno];
            systemq.ending = pcb[portno];
            pcb[portno].next_cb = pcb[portno];
            pcb[portno].previous_cb = pcb[portno];*/
          end  /* if */
         else
          begin
            /*pcb[portno].next_cb = systemq.start;
            pcb[portno].previous_cb = systemq.ending.next_cb;
            systemq.ending.next_cb = pcb[portno];
            systemq.ending = pcb[portno];
            systemq.start.previous_cb = system.ending;*/
          end   /* else */
         systemq.qcount += 1;
         break;
```

```
        end   /* case 0 */

      case 1:

/* insert into the readylq  */

        begin
          if (readylq.qcount == 0)
           begin
             /*readylq.start = pcb[portno];
             readylq.ending = pcb[portno];
             pcb[portno].next_cb = pcb[portno];
             pcb[portno].previous_cb = pcb[portno];*/
           end  /* if */
          else
           begin
             /*pcb[portno].next_cb = readylq.start;
             pcb[portno].previous_cb = readyq.ending;
             readylq.ending.next_cb = pcb[portno];
             readylq.ending = pcb[portno];
             readylq.start.previous_cb = readylq.ending;*/
           end  /* else */
          readylq.qcount += 1;
          break;
        end   /* case 1 */

  /*  case 2:   this will be for expansion, ie.  */
  /*            if another ready queue (ready2q) */
  /*            were needed and implemented      */
    end /* switch's begin */

 else
  begin

/* insert into the iowaitq  */

    if (iowaitq.qcount == 0)
     begin
       /*iowaitq.start = pcb[portno];
       iowaitq.ending = pcb[portno];
       pcb[portno].next_cb = pcb[portno];
       pcb[portno].previous_cb = pcb[portno];*/
     end  /* if */

    else
     begin
/*       pcb[portno].next_cb = iowaitq.start;
       pcb[portno].previous_cb = iowaitq.ending;
       iowaitq.ending.next_cb = pcb[portno];
       iowaitq.ending = pcb[portno];
       iowaitq.start.previous_cb = pcb[portno];*/
     end  /* else */
```

```
/*****************************************************************/
/*                                                               */
/*                      PROCESS_SCHEDULER                        */
/*                                                               */
/*     Date:  11 October 1983                                    */
/*     Version:   1.1                                            */
/*                                                               */
/*     Name:   process_scheduler                                 */
/*     Module Number: 1.2.3.5.2.2                                */
/*     Function:       To process the I/O wait queue, get the    */
/*                     get the next ready process by checking    */
/*                     the system queue for a process, then if   */
/*                     none check the ready queues. If one       */
/*                     of the queues has a read  rocess then     */
/*                     the process is taken of  he queue and     */
/*                     executed.                                 */
/*                                                               */
/*     Calling Modules:  execute_command                         */
/*     Modules Called:   send_file, write, pro  f  iowaitq,      */
/*                       check_readyqs, get_pcb, rogram_run,     */
/*                       run_sys_comm, deallocate_space          */
/*                                                               */
/*     Global Variables Used:  portno                            */
/*     Global Variables Changed:  portno                         */
/*                                                               */
/*     Author:  Paul E. Cruser                                   */
/*     System:  VAX 11/780, VMS O/S and UNIX O/S: testing only   */
/*                                                               */
/*****************************************************************/

process_scheduler()
begin
#define    command_type_error     10 /* code for error received */
int next_process;
   process_iowaitq();
   next_process = check_readyqs();
   switch (next_process)
    begin
    case -1:
      return(1);

    case 0:
      begin
       get_pcb(next_process);
       run_sys_comm();
       return(1);
      end

    case 1:
/*  case 2:   */
/*  etc...    */
      begin
```

```
            get_pcb(next_process);
            switch(pcb[portno].command_type)
             begin
               case 0: { program_run(); break; }
               case 1:
               case 4: { send_file(portno); break; }
               case 2: { send_file(noports); break; }
               case 3: { write(portno); break; }
               default: { error(command_type_error);  return(0);}
             end  /* switch */
           deallocate_space(portno);
           return(1);
           end
       end
end
```

```
/*****************************************************************/
/*                                                               */
/*                        CHECK_READYQS                          */
/*                                                               */
/*     Date:  14 September 1983                                  */
/*     Version:  1.0                                             */
/*                                                               */
/*     Name:  check_readyqs                                      */
/*     Module Number: 1.2.3.5.2.2.2.1                            */
/*     Function:    To check all the ready queues to see if      */
/*                  there are any processes on them.  It checks  */
/*                  the queues in order of priority and when     */
/*                  the first non-empty queue is found its       */
/*                  priority is sent back to the caller.  If     */
/*                  all of the queues are empty the value -1 is  */
/*                  returned.                                    */
/*                                                               */
/*     Calling Modules: process_scheduler                        */
/*     Modules Called:  none                                     */
/*                                                               */
/*     Global Variables Used: pcb[], systemq, readylq            */
/*     Global Variables Changed: none                            */
/*                                                               */
/*     Author:  Paul E. Cruser                                   */
/*     System:  VAX 11/780, VMS O/S and UNIX O/S: testing only   */
/*                                                               */
/*****************************************************************/

check_readyqs()
begin
  if (systemq.qcount > 0)
     return(0); /* systemq has at least one process */
  else
   if (readylq.qcount > 0)
     return(1); /* readylq has at least one process */

/* else                                */
/*  if (ready2q.qcount > 0)            */
/*    return(2);                       */
/*   etc...                            */
   else
    return(-1);  /* all the queues are empty */
end
```

```
/*****************************************************************/
/*                                                               */
/*                          GET_PCB                              */
/*                                                               */
/*     Date:  15 September 1983                                  */
/*     Version: 1.1                                              */
/*                                                               */
/*     Name:  get_pcb                                            */
/*     Module Number: 1.2.3.5.2.2.2                              */
/*     Function:    To get the next pcb and take it off the      */
/*                  queue which it is on.  The next pcb is the   */
/*                  first one on the queue.                      */
/*                                                               */
/*     Calling Modules:  process_scheduler                       */
/*     Modules Called:   none                                    */
/*                                                               */
/*     Global Variables Used:  systemq, readylq, portno          */
/*     Global Variables Changed: systemq, readylq, portno        */
/*                                                               */
/*     Author:  Paul E. Cruser                                   */
/*     System:  VAX 11/780, VMS O/S and UNIX O/S: testing only   */
/*                                                               */
/*****************************************************************/

get_pcb(tyrone)
int tyrone;
begin
int i;
struct z8pcb *temp;
   switch(tyrone)
    begin
      case 0:
       begin
         portno = systemq.start.port_of_origin;
         if (systemq.qcount == 1)
            systemq.qcount -= 1;
         else
          begin
            temp = systemq.start;
            systemq.start = temp.next_cb;
            systemq.start.previous_cb =
                     temp.previous_cb.previous_cb;
            systemq.qcount -= 1;
          end
         break;
       end /* case 0 */

      case 1:
       begin
         portno = readylq.port_of_origin;
         if (readylq.qcount == 1)
          begin
```

```
                readylq.start = 0;
                readylq.ending = 0;
                readylq.qcount -=1;
              end
             else
              begin
                temp = readylq.start;
                readylq.start = temp.next_cb;
                readylq.previous_cb =
                      temp.previous_cb.previous_cb;
                readylq.qcount -= 1;
              end
             break;
           end  /* case 1 */
       end /* switch */
end
```

```
/*****************************************************************/
/*                                                               */
/*                      PROCESS_IOWAITQ                          */
/*                                                               */
/*     Date:  15 September 1983                                  */
/*     Version:   1.0                                            */
/*                                                               */
/*     Name:   process_iowaitq                                   */
/*     Module Number: 1.2.3.5.2.2.1                              */
/*     Function:    To transfer any process on the wait queue    */
/*                  that is done with i/o wait to a ready queue  */
/*                                                               */
/*     Calling Modules:  process_scheduler                       */
/*     Modules Called:   insert_pcb                              */
/*                                                               */
/*     Global Variables Used:  iowaitq, noports, portno          */
/*     Global Variables Changed: iowaitq                         */
/*                                                               */
/*     Author:  Paul E. Cruser                                   */
/*     System:  VAX 11/780, VMS O/S and UNIX O/S: testing only   */
/*                                                               */
/*****************************************************************/

process_iowaitq()
begin
int  i,tempportno;
struct z8pcb *temp;
   if (iowaitq.qcount > 0)
    begin
      temp = iowaitq.start;
      i = 0;
      while (i < noports)
       begin
         if (temp.io_status == 0)
          begin
            tempportno = portno;
            portno = temp.port_of_origin;
            temp.next_cb.previous_cb = temp.previous_cb;
            temp.previous_cb.next_cb = temp.next_cb;
            iowaitq.qcount -= 1;
            insert_pcb();
            portno = tempportno;
            return(1);
          end   /* if */
         i += 1;
       end  /* while */
     return(1);
    end  /* if */
end  /* process_iowaitq */
```

```
/*****************************************************************/
/*                                                               */
/*                        CLEAN_UP                               */
/*                                                               */
/*     Date:  15 September 1983                                  */
/*     Version:  1.0                                             */
/*                                                               */
/*     Name:  clean_up                                           */
/*     Module Number: 00                                         */
/*     Function:     This subroutine will delete any process     */
/*                   from the queue it resides in and end it     */
/*                   without finishing.  It is not used in this  */
/*                   implementation, but can be useful later.    */
/*                                                               */
/*     Calling Modules: none                                     */
/*     Modules Called: deallocate_space                          */
/*                                                               */
/*     Global Variables Used: pcb[], portno, iowaitq, systemq    */
/*                            readylq                            */
/*     Global Variables Changed: iowaitq, systemq, readylq,      */
/*                               pcb[]                           */
/*                                                               */
/*     Author:  Paul E. Cruser                                   */
/*     System:  VAX 11/780, VMS O/S and UNIX O/S: testing only   */
/*                                                               */
/*****************************************************************/

clean_up()
begin
   pcb[portno].next_cb.previous_cb = pcb[portno].previous_cb;
   pcb[portno].previous_cb.next_cb = pcb[portno].next_cb;
   deallocate_space(portno);
   switch (pcb[portno].current_q)
    begin
      case -1: { iowaitq.qcount -= 1; return(1); }
      case  0: { systemq.qcount -= 1; return(1); }
      case  1: { readylq.qcount -= 1; return(1); }
   /* case  2: { ready2q.qcount -= 1; return(1); }  */
   /* etc...                                        */
      default: return(0);
    end  /* switch */
end  /* clean_up */
```

```
/*****************************************************************/
/*                                                               */
/*                        RUN_SYS_COMM                           */
/*                                                               */
/*     Date:   15 September 1983                                 */
/*     Version:  1.0                                             */
/*                                                               */
/*     Name:   run_sys_comm                                      */
/*     Module Number: 1.2.3.4.2                                  */
/*     Function:     To run the system command                   */
/*                                                               */
/*     Calling Modules:  process_scheduler                       */
/*     Modules Called:   none                                    */
/*                                                               */
/*     Global Variables Used: none                               */
/*     Global Variables Changed: none                            */
/*                                                               */
/*     Author:  Paul E. Cruser                                   */
/*     System:  VAX 11/780, VMS O/S and UNIX O/S: testing only   */
/*                                                               */
/*****************************************************************/

run_sys_comm()
begin
/*   The system commands are run in the module system_change     */
/*  and do not need to be run here.  This module will be needed  */
/*  when the system_change will be actually implemented on the   */
/*  host micro-system.                                           */
   return(1);
end
```

```
/*****************************************************************/
/*                                                               */
/*                          PROGRAM_RUN                          */
/*                                                               */
/*     Date:  15 September 1983                                  */
/*     Version: 1.0                                              */
/*                                                               */
/*     Name:   program_run                                       */
/*     Module Number: 1.2.3.5.2.2.3.4                            */
/*     Function:   This routine will call a Z800 Assembly sub-   */
/*                 routine that will enable the interrupts and   */
/*                 start execution of the file pointed to by     */
/*                 pcb's offset address.                         */
/*                                                               */
/*     Calling Modules:  process_scheduler                       */
/*     Modules Called:   amoskernel                              */
/*                                                               */
/*     Global Variables Used:  pcb[], portno                     */
/*     Global Variables Changed: none                            */
/*                                                               */
/*     Author:  Paul E. Cruser                                   */
/*     System:  VAX 11/780, VMS O/S and UNIX O/S: testing only   */
/*                                                               */
/*****************************************************************/

program_run()
begin
   if (amoskernel(pcb[portno].offset_address,
                          pcb[portno].final_address))
     return(1);
   else
     return(0);
end
```

```
/*****************************************************************/
/*                                                               */
/*                        DET_VALID_COMM                         */
/*                                                               */
/*     Date: 1 Sept 1983                                         */
/*     Version: 1.2                                              */
/*                                                               */
/*     Name: det_valid_comm                                      */
/*     Module Number: 1.2.3                                      */
/*     Function: This module will detrmine the command the user  */
/*               is requesting then will call the necessary      */
/*               routines to have the specific command executed. */
/*                                                               */
/*     Calling Modules: main                                     */
/*     Modules Called: log_in_user, stringcmp, log_out_user,     */
/*                     help_user, system_change, user_command,   */
/*                     execute_user_command,error                */
/*                                                               */
/*     Global Variables Used: command_table.command              */
/*     Global Variables Changed: None                            */
/*                                                               */
/*     Author: Ronald K. Miller                                  */
/*     System: VAX 11/780, VMS O/S and UNIX O/S: testing only    */
/*                                                               */
/*****************************************************************/

det_valid_comm()
begin
   char *bye,*help,*system;    /* Initialize comparision string */
   bye = "BYE";
   help = "HELP";
   system = "SYS";
   if(log_in_user() != 2) return(1);
    if(stringcmp(bye,command_table.command))
    begin
      log_out_user();
      return(1);
    end
    if(stringcmp(help,command_table.command))
    begin
      help_user();
      return(1);
    end
    if(stringcmp(system,command_table.command))
    begin
      system_change();
      return(1);
    end
```

```
        if(user_command())
        begin
          execute_user_command();
          return(1);
        end
      error(1);
end
```

```
/*****************************************************************/
/*                                                               */
/*                          USER_COMMAND                         */
/*                                                               */
/*     Date: 1 Sept 1983                                         */
/*     Version: 1.0                                              */
/*                                                               */
/*     Name: user_command                                        */
/*     Module Number: 1.2.3B                                     */
/*     Function: This module determines if the command           */
/*               requested is a user command.                    */
/*                                                               */
/*     Calling Modules: det_valid_comm                           */
/*     Modules Called: stringcmp                                 */
/*                                                               */
/*     Global Variables Used:  command_table.command, cmd        */
/*     Global Variables Changed: cmd                             */
/*                                                               */
/*     Author: Ronald K. Miller                                  */
/*     System: VAX 11/780, VMS O/S and UNIX O/S : testing only   */
/*                                                               */
/*****************************************************************/

user_command()
begin
   char *run,*list,*print,*delete,*directory;
                                        /* Initialize comparision */
                                        /* strings                */
   run = "RUN";list = "LIST"; print = "PRINT";
   delete = "DEL";directory = "DIR";
   if(stringcmp(run,command_table.command))
   begin
     cmd = 1; return(1);           /* cmd=1 implies run command */
   end
   if(stingcmp(list,command_table.command))
   begin
     cmd = 2; return(1);          /* cmd=2 implies list command */
   end
   if(stringcmp(print,command_table.command))
   begin
     cmd = 3; return(1);         /* cmd=3 implies print command */
   end
   if(stringcmp(delete,command_table.command))
   begin
     cmd = 4; return(1);        /* cmd=4 implies delete command */
   end
```

```
    if(stringcmp(directory,command_table.command))
    begin
      cmd = 5; return(1);      /* cmd=5 implies directory command */
    end
    return(0);                    /* return(0) implies not a user */
                                  /* command                      */
end
```

```
/*****************************************************************/
/*                                                               */
/*                          STRINGCMP                            */
/*                                                               */
/*     Date: 1 Sept 1983                                         */
/*     Version: 1.0                                              */
/*                                                               */
/*     Name: stringcmp                                           */
/*     Module Number: 1.2.3A                                     */
/*     Function: Determines lexicographic equality of two        */
/*               strings.                                        */
/*                                                               */
/*     Calling Modules: det_valid_command, user_command          */
/*     Modules Called: None                                      */
/*                                                               */
/*     Global Variables Used: None                               */
/*     Global Variables Changed: None                            */
/*                                                               */
/*     Author: Ronald K. Miller                                  */
/*     System: VAX 11/780, VMS O/S and UNIX O/S: testing only    */
/*                                                               */
/*****************************************************************/

stringcmp(s,t)
  char s[],t[];
  begin
    int i,j;
    i = j = 0;
    while (s[i++] == t[j++])
                              /* Continue while strings are equal */
                              /* Return(1) if strings are equal   */
      if(s[i] == '\0') return(1);
    return(0);
  end
```

```
/*****************************************************************/
/*                                                               */
/*                      EXECUTE_USER_COMMAND                     */
/*                                                               */
/*     Date: 1 Sept 1983                                         */
/*     Version: 1.0                                              */
/*                                                               */
/*     Name: execute_user_command                                */
/*     Module Number: 1.2.3.5                                    */
/*     Function: This module will have the user command checked  */
/*               for validity and if valid will have it          */
/*               executed.                                       */
/*                                                               */
/*     Calling Modules: det_valid_comm                           */
/*     Modules Called: validate_user_command, execute_command    */
/*                                                               */
/*     Global Variables Used: None                               */
/*     Global Variables Changed: None                            */
/*                                                               */
/*     Author: Ronald K. Miller                                  */
/*     System: VAX 11/780, VMX O/S and UNIX O/S: testing only    */
/*                                                               */
/*****************************************************************/

execute_user_command()
begin
   if(validate_user_command())
   begin
     execute_command();return(1);
   end
   return(0);
end
```

```
/*****************************************************************/
/*                                                               */
/*                    VALIDATE_USER_COMMAND                      */
/*                                                               */
/*     Date: 1 Sept 1983                                         */
/*     Version: 1.2                                              */
/*                                                               */
/*     Name: validate_user_command                               */
/*     Module Number: 1.2.3.5.1                                  */
/*     Function: This module will check to see if the proper     */
/*               file is being requested by the authorized user  */
/*               and if the files are located on the disk.  In   */
/*               the case of a RUN command this module will      */
/*               also check to see if the requested file is an   */
/*               executable file.                                */
/*                                                               */
/*     Calling Modules: execute_user_command                     */
/*     Modules Called: chk_run_file, chk_filename,               */
/*                     chk_user_name                             */
/*                                                               */
/*     Global Variables Used: cmd                                */
/*     Global Variables Changed: None                            */
/*                                                               */
/*     Author: Ronald K. Miller                                  */
/*     System: VAX 11/780, VMS O/S and UNIX O/S: testing only    */
/*                                                               */
/*****************************************************************/

validate_user_command()
begin
  switch(cmd)
  begin
   case 1:{if(chk_filename() && chk_user_name() && chk_run_file())
               return(1);
           return(0);}
                              /* return(1) if all three are valid */
                              /* otherwise will return(0)         */
   case 2: case 4: {if(chk_filename() && chk_user_name())
                        return(1);
                    return(0);}
   case 3: {if(chk_filename() && chk_user_name() &&
                !chk_run_file()) return(1);
            return(0);}
   case 5: return(1);  /* directory command automatically valid */
   default: return(0);
  end
end
```

```
/*****************************************************************/
/*                                                               */
/*                          CHK_FILENAME                         */
/*                                                               */
/*     Date: 1 Sept 1983                                         */
/*     Version: 1.0                                              */
/*                                                               */
/*     Name: chk_filename                                        */
/*     Module Number: 1.2.3.5.1.1.1                              */
/*     Function: This module will check to see if the file       */
/*               being requested is located on the disk.         */
/*                                                               */
/*     Calling Modules: validate_user_command                    */
/*     Modules Called: open, error, close                        */
/*                                                               */
/*     Global Variables Used:  command_table.parameter1          */
/*     Global Variables Changed: None                            */
/*                                                               */
/*     Author: Ronald K. Miller                                  */
/*     System: VAX 11/780, VMS O/S and UNIX O/S: testing only    */
/*                                                               */
/*****************************************************************/

chk_filename()
begin
  if(fd=(open(command_table.parameter1,0)) == ERROR)
  begin
    error(2);                    /* if file can not be open the  */
    return(0);                   /* file is not on the disk      */
  end
  else
  begin
    get_username(fd);
    close(fd);              /* else close the file and return(1) */
    return(1);
  end
end
```

```
/*****************************************************************/
/*                                                               */
/*                          CHK_RUN_FILE                         */
/*                                                               */
/*     Date: 5 Sept 1983                                         */
/*     Version: 1.0                                              */
/*                                                               */
/*     Name: chk_run_file                                        */
/*     Module Number: 1.2.3.5.1.1.3                              */
/*     Function: This module is to determine if the file being   */
/*               requested is an executable file.  Will          */
/*               return(1) if so otherwise will return(0).       */
/*                                                               */
/*     Calling Modules: validate_user_command                    */
/*     Modules Called: stringcmp                                 */
/*                                                               */
/*     Global Variables Used: command_table.parameter1           */
/*     Global Variables Changed: None                            */
/*                                                               */
/*     Author: Ronald K. Miller                                  */
/*     System: VAX 11/780, VMS O/S and UNIX O/S: testing only    */
/*                                                               */
/*****************************************************************/

chk_run_file()
begin
   char temp[3],*com;
   int i,j;
   i = j = 0;
   com = "COM";
   while(command_table.parameter1[i++] != ".");
                              /* Find the extension part of the  */
   while (j < 3)              /* filename                        */
     temp[j++]= command_table.parameter1[i++];
                              /* Place the extension in a        */
   if(stringcmp(com,temp))    /* temporary buffer for comparsion */
   begin
     if(cmd == 3) error(11); /* error(11) means an executable   */
     return(1);               /* file is trying to be ran        */
   end
   if(cmd == 1) error(10);    /* error(10) means a nonexecutable */
   return(0);                 /* file is trying to be printed    */
end
```

```
/*****************************************************************/
/*                                                               */
/*                          CHK_USER_NAME                        */
/*                                                               */
/*     Date: 5 Sept 1983                                         */
/*     Version: 1.0                                              */
/*                                                               */
/*     Name: chk_user_name                                       */
/*     Module Number: 1.2.3.5.1.1.2                              */
/*     Function: This module will check to make sure that the    */
/*               proper user is requesting their own file.       */
/*                                                               */
/*     Calling Modules: validate_user_command                    */
/*     Modules Called: stringcmp                                 */
/*                                                               */
/*     Global Variables Used: userblocks[portno].usernm,         */
/*                            file_username                      */
/*     Global Variables Changed: None                            */
/*                                                               */
/*     Author: Ronald K. Miller                                  */
/*     System: VAX 11/780, VMS O/S and UNIX O/S: testing only    */
/*                                                               */
/*****************************************************************/

chk_user_name()
begin
   if(stringcmp(file_username,userblocks[portno].usernm))
     return(1);
   error(3);
   return(0);
end
```

```
/*****************************************************************/
/*                                                               */
/*                          GET_USERNAME                         */
/*                                                               */
/*     Date: 5 Sept 1983                                         */
/*     Version: 1.0                                              */
/*                                                               */
/*     Name: get_username                                        */
/*     Module Number: 1.2.3.5.1.1.1.2                            */
/*     Function: This module will get the user name of the file  */
/*               being requested.  This is in order to check     */
/*               for user authority in a later module.           */
/*                                                               */
/*     Calling Modules: chk_filename                             */
/*     Modules Called: None                                      */
/*                                                               */
/*     Global Variables Used: opened_files, file_username,       */
/*                            BEGINUSER, ENDUSER                 */
/*     Global Variables Changed: file_username                   */
/*                                                               */
/*     Author: Ronald K. Miller                                  */
/*     System: VAX 11/780, VMS O/S and UNIX O/S: testing only    */
/*                                                               */
/*****************************************************************/

get_username(fd)
begin
   int i,j;
   j=BEGINUSER;i=0;
   while(j < ENDUSER+1)
     file_username[i++] = opened_files[fd,j++];
   return;
end
```

```
/*****************************************************************/
/*                                                               */
/*                            OPEN                               */
/*                                                               */
/*     Date: 6 Sept 1983                                         */
/*     Version: 1.3                                              */
/*                                                               */
/*     Name: open                                                */
/*     Module Number: 1.2.3.5.1.1.1.1.1                          */
/*     Function: This module opens a file from disk and then     */
/*               allows for reading from and writing to the      */
/*               particular file.                                */
/*                                                               */
/*     Calling Modules: chk_filename                             */
/*     Modules Called: get_directory                             */
/*                                                               */
/*     Global Variables Used: file_username, opened_files, name, */
/*                            NUMSEC, BUFLENGTH, BUFSIZE,        */
/*                            DIRTRACK, DIRSECTOR, file, dtrack, */
/*                            dsector, del_track, del_sector     */
/*     Global Variables Changed: file_username, opened_file      */
/*                                                               */
/*     Author: Ronald K. Miller                                  */
/*     System: VAX 11/780, VMS O/S and UNIX O/S: testing only    */
/*                                                               */
/*****************************************************************/

open(file,code)
  char file[];
  int code;
begin
   int i,j,k,location;
   char buffer[BUFSIZE] [BUFLENGTH];
   dtrack=DIRTRACK;dsector=DIRSECTOR;
   finished = 0;
   while (!finished)
   begin
     read(dtrack,dsector,buffer);
                          /* Reads in a block of the directory */
                          /* and places it into buffer.        */
     i=0;
     while ((i < BUFLENGTH) && (buffer[i,0] != EOF))
                          /* Check of end of buffer or the end */
                          /* of the directory.                 */
     begin
       k=0;
       while (k < NAMESIZE)
       begin
         name[k]=buffer[i] [k];/* Places each filename into name*/
         k++;                  /* for comparision.              */
       end
       if(stringcmp(name,file))  /* If the names are the same   */
```

```
            begin                          /* Place into opened_file      */
              j = 0;                       /* and then return(l).         */

              del_track = dtrack;          /* Track and sector of where   */
              del_sector = dsector;        /* the filename is located     */
                                           /* in the directory.           */
              while (j < BUFSIZE)
              begin
                opened_files[location] [j] = buffer[i] [j];
                j++;
              end
              location++;
              return(location - 1);
            end
            i++;
          end
          if (i = BUFLENGTH)
                              /* If true means the entire directory    */
          begin               /* has not be read. Therefore another    */
            dsector++;        /* sector needs to be read in. If        */
                              /* the last sector on the track has been */
                              /* need to go to next track and sector 0.*/
          if (dsector > NUMSEC)
            begin
              dtrack++;
              dsector = 0;
            end
          end
          else return(-1);    /* return(-1) means reached end of the   */
                              /* directory and no file was found.      */
      end
    end
```

```
/*****************************************************************/
/*                                                               */
/*                            ERROR                              */
/*                                                               */
/*     Date: 6 Sept 1983                                         */
/*     Version: 1.5                                              */
/*                                                               */
/*     Name: error                                               */
/*     Module Number: 1.2.3.5.1.1.1.3                            */
/*     Function: This module will determine error received and   */
/*               will build the necessary error message.         */
/*                                                               */
/*     Calling Modules: det_valid_comm, chk_filename,            */
/*                      chk_user_name, execute_command,          */
/*                      chk_space                                */
/*     Modules Called:  transmit_message                         */
/*                                                               */
/*     Global Variables Used: message, MESSIZE                   */
/*     Global Variables Changed: message                         */
/*                                                               */
/*     Author: Ronald K. Miller                                  */
/*     System: VAX 11/780, VMS O/S and UNIX O/S: testing only    */
/*                                                               */
/*****************************************************************/

error(type)
int type;
begin
   switch (type)
   begin
     case 1: { strcpy(message,
               "SYNTAX ERROR                                \n");
               break;}
     case 2: { strcpy(message,
               "INVALID FILENAME                            \n");
               break;}
     case 3: { strcpy(message,
               "INVALID USER ATTEMPTING TO RETRIVE FILE\n");
               break;}
     case 4: { strcpy(message,
               "ILLEGAL USER                                \n");
               break;}
     case 5: { strcpy(message,
               "UNAUTHORIZED USER                           \n");
               break;}
     case 6: { strcpy(message,
               "UNRECONIZEABLE CODE - TRY AGAIN             \n");
               break;}
     case 7: { strcpy(message,
               "NOT ENOUGH MEMORY SPACE TO EXECUTE NOW \n");
               break;}
     case 8: { strcpy(message,
```

```
                "PROGRAM TOO LARGE FOR MEMORY          \n");
                break;}
        case 9: { strcpy(message,
                "PROCESS ABORTED, DID NOT COMPLETE...  \n");
                break;}
        case 10:{ strcpy(message,
                "NONEXECUTABLE FILE, UNABLE TO RUN     \n");
                break;}
        case 11:{ strcpy(message,
                "EXECUTABLE FILE, UNABLE TO PRINT      \n");
                break;}
        default: return(0);
      end
      transmit_message(message);
      return(0);
end
```

```
/*****************************************************************/
/*                                                               */
/*                        TRANSMIT_MESSAGE                       */
/*                                                               */
/*     Date: 7 Sept 1983                                         */
/*     Version: 1.0                                              */
/*                                                               */
/*     Name: transmit_message                                    */
/*     Module Number: 1.2.3.1.2.1                                */
/*     Function: This module is to transmit any message          */
/*               received to the correct user.                   */
/*                                                               */
/*     Calling Modules: error, build_message                     */
/*     Modules Called: None                                      */
/*                                                               */
/*     Global Variables Used: None                               */
/*     Global Variables Changed: None                            */
/*                                                               */
/*     Author: Ronald K. Miller                                  */
/*     System: VAX 11/780, VMS O/S and UNIX O/S: testing only    */
/*                                                               */
/*****************************************************************/

transmit_message(string)
  char string[];
begin
   int i;
   i=0;
   do begin
     while ((inp(ports[portno].statport) &
                 ports[portno].sendbit) == 0);
     outp(ports[portno].dataport,string[i]);
   end
   while (string[i++] != '\n');
   return;
end
```

```
/*****************************************************************/
/*                                                               */
/*                            READ                               */
/*                                                               */
/*     Date: 7 Sept 1983                                         */
/*     Version: 1.2                                              */
/*                                                               */
/*     Name: read                                                */
/*     Module Number: 1.2.3.5.2.1.3                              */
/*     Function: To read a sector from the disk when given the   */
/*               track number and sector number to read.         */
/*                                                               */
/*     Calling Modules: open                                     */
/*     Modules Called: None                                      */
/*                                                               */
/*     Global Variables Used: None                               */
/*     Global Variables Changed: None                            */
/*                                                               */
/*     Author: Ronald K. Miller                                  */
/*     System: VAX 11/780, VMS O/S and UNIX O/S: testing only    */
/*                                                               */
/*****************************************************************/

read(track,sector,inbuf)
  int track,sector;
  char inbuf[BUFSIZE] [BUFLENGTH];
begin
   int k,i,j;
   char c;
   i = j = k = 0;
   while (k < BYTE_SIZE)
   begin
     while ((inp(DISKSTAT) & DISKRDA) == 0);
     c = inp(DISKPORT);
     inbuf[i] [j] = c;j++;
     if (c == EOF) finished = 1; /* When EOF reached the entire */
     if (j == BUFLENGTH)         /* file has been read in.      */
     begin
       i++; j = 0;
     end
     k++;
   end
   return;
end
```

```
/*****************************************************************/
/*                                                               */
/*                        BUILD_PARSE_TABLE                      */
/*                                                               */
/*     Date: 7 Sept 1983                                         */
/*     Version: 1.4                                              */
/*                                                               */
/*     Name: build_parse_table                                   */
/*     Module Number: 1.2.2.3                                    */
/*     Function: This module will break the command line into    */
/*               its different parameters.                       */
/*                                                               */
/*     Calling Modules: p_comm_line                              */
/*     Modules Called: None                                      */
/*                                                               */
/*     Global Variables Used: command_line, command_table        */
/*     Global Variables Changed: command_table                   */
/*                                                               */
/*     Author: Ronald K. Miller                                  */
/*     System: VAX 11/780, VMS O/S and UNIX O/S: testing only    */
/*                                                               */
/*****************************************************************/

build_parse_table()
begin
#define BLANK    ' '   /* defines a blank */
   int i,j,k,l,m;
   i = j = k = l = m = 0;
   while (l < COMMSIZE)
   begin
      command_table.command[l] = BLANK;
      command_table.parameter1[l]=
                     command_table.parameter2[l] = BLANK;
      l++;
   end
   while (l < PARASIZE)
   begin
      command_table.parameter1[l]=
                     command_table.parameter2[l] = BLANK;
      l++;
   end
   while (command_line[i] != BLANK && command_line[i] != '\n')
     command_table.command[j++] = command_line[i++];
   while (command_line[i++] == BLANK);
   if(command_line[i] == '\n')
   begin
     command_table.numparam = 0;
     return;
   end
   while (command_line[i] != BLANK && command_line[i] != '\n')
     command_table.parameter1[k++] = command_line[i++];
   while (command_line[i++] == BLANK);
```

```
    if(command_line[i] == '\n')
    begin
      command_table.numparam = 1;
      return;
    end
    while (command_line[i] != BLANK && command_line[i] != '\n')
      command_table.parameter2[m++] = command_line[i++];
    command_table.numparam = 2;
    return;
end
```

```
/*****************************************************************/
/*                                                               */
/*                        EXCEUTE_COMMAND                        */
/*                                                               */
/*     Date: 9 Sept 1983                                         */
/*     Version: 2.1                                              */
/*                                                               */
/*     Name: execute_command                                     */
/*     Module Number: 1.2.3.5.2                                  */
/*     Function: This module will execute a given user command   */
/*               only after the command has been determined to   */
/*               be valid.                                       */
/*                                                               */
/*     Calling Modules: execute_user_command                     */
/*     Modules Called: get_file, send_file, process_scheduler    */
/*                                                               */
/*     Global Variables Used: cmd                                */
/*     Global Variables Changed: None                            */
/*                                                               */
/*     Author: Ronald K. Miller                                  */
/*     System: VAX 11/780, VMS O/S and UNIX O/S: testing only    */
/*                                                               */
/*****************************************************************/

execute_command()
begin
   switch(cmd)
   begin
     case 1:
     case 2:
     case 3: { if(get_file(atoi(opened_files[fd,BEGTRACK]),
                           atoi(opened_files[fd,BEGSECTOR])))
                  process_scheduler(); break; }
     case 4: { if(get_file(del_track,del_sector))
                  process_scheduler(); break; }
     case 5: { if(get_file(DIRTRACK,DIRSECTOR))
                  process_scheduler(); break; }
     default: { error(6);break; }
   end
   return;
end
```

```
/*****************************************************************/
/*                                                               */
/*                          GET_FILE                             */
/*                                                               */
/*     Date: 9 Sept 1983                                         */
/*     Version: 1.2                                              */
/*                                                               */
/*     Name: get_file                                            */
/*     Module Number: 1.2.3.5.2.1                                */
/*     Function: This module preforms all the necessary          */
/*               functions to run an executable file.            */
/*                                                               */
/*     Calling Modules: execute_command, build_pcb               */
/*     Modules Called: read, chk_space                           */
/*                                                               */
/*     Global Variables Used: opened_files, BEGSIZE, size,       */
/*                            begin_address, end_address         */
/*     Global Variables Changed: begin_address, end_address      */
/*                                                               */
/*     Author: Ronald K. Miller                                  */
/*     System: VAX 11/780, VMS O/S and UNIX O/S: testing only    */
/*                                                               */
/*****************************************************************/

get_file(track,sector)
  int track,sector;
begin
   int i,j;
   char charsize;
   charsize = opened_files[fd,BEGSIZE];
   size = atoi(charsize);      /* Converting a string to integer */
   if (!chk_space(size))       /* to check if enough space       */
   begin
     error(7);                 /* Not enough space to put the    */
     return(0);                /* program into main memory       */
   end
   build_pcb(cmd-1);
   finished = 0;
   begin_address[number_jobs] = memory_loc;
                               /* this sets the first location   */
                               /* where the program is being     */
                               /* placed                         */
   pcb[portno].offset_address = memory_loc;
   track = opened_files[fd,BEGTRACK];
   sector = opened_files[fd,BEGSECTOR];
   while(!finished)
   begin
     read(track,sector,memory_loc);
     memory_loc = memory_loc  + BYTE_SIZE;
     track = memory_loc - 2;
     sector = memory_loc - 1;
     if(track == sector == 0)
```

```
        finished = 1;
      if(cmd == 4)
                finished= 1; /* If delete command only need to  */
                             /* read one sector from the disk.  */
      memory_loc = memory_loc - 2;
                             /* Don't want to have the next     */
                             /* track and sector in memory.     */
    end
    end_address[number_jobs] = memory_loc - BYTE_SIZE;
                             /* This sets the last location     */
                             /* where the program is begin      */
                             /* placed.                         */
    pcb[portno].final_address = memory_loc - BYTE_SIZE;
    pcb[portno].io_status = 0;
                        /* The process is no longer in i/o wait    */
    number_jobs++;      /* Total number of jobs in the system.     */
end
```

```
/*****************************************************************/
/*                                                               */
/*                          CHK_SPACE                            */
/*                                                               */
/*     Date: 13 Sept 1983                                        */
/*     Version: 1.2                                              */
/*                                                               */
/*     Name: chk_space                                           */
/*     Module Number: 1.2.3.5.2.1.1                              */
/*     Function: This module will determine if there is enough   */
/*               space in main memory to place the incoming      */
/*               program.                                        */
/*                                                               */
/*     Calling Modules: get_file                                 */
/*     Modules Called: sort                                      */
/*                                                               */
/*     Global Variables Used: size, begin_address, end_address,  */
/*                            order, number_jobs, BASE_ADDRESS,  */
/*                            TOP_ADDRESS                        */
/*     Global Variables Changed:  None                           */
/*                                                               */
/*     Author: Ronald K. Miller                                  */
/*     System: VAX 11/780, VMS O/S and UNIX O/S: testing only    */
/*                                                               */
/*****************************************************************/

chk_space(size)
  int size;
begin
  int i,j,k,l,bottom;
  i = j = k = 0;
  if(number_jobs>1) /* If the more than one job on the system   */
    sort(order);    /* the location must be sorted from largest */
                    /* to smallest.  The sorted order is placed */
                    /* in the array called order.               */
  else if(number_jobs == 0)
    order[0] = MAXJOBS;
                    /* If no jobs on the system order must be   */
                    /* initialized and this make the first value*/
                    /* equal to TOP_ADDRESS.                    */
  else
  begin
    order[0] = 0;   /* If only one job order must be inialized. */
    order[1] = MAXJOBS;
  end
  i = order[j];
  bottom = BASE_ADDRESS;
                    /* bottom is use to find how much free      */
                    /* space is located between jobs.           */
  while(k < number_jobs+1)
  begin
    if(size <= (begin_address[i]-bottom))
```

```
      begin
        memory_loc = bottom;
        return(1);
                        /* Make memory_loc equal to the last entry  */
                        /* between jobs and then return(1).         */
      end
      if(k == 0)
      begin
        error(8);
        return(0);
      end
      bottom = end_address[i];
                        /* Move bottom to the last location of the  */
                        /* next job.                                */
      i = order[j++];
      k++;
    end
    error(7);
    return(0);
  end
```

```
/*****************************************************************/
/*                                                               */
/*                          SORT                                 */
/*                                                               */
/*     Date: 13 Sept 1983                                        */
/*     Version: 1.1                                              */
/*                                                               */
/*     Name: sort                                                */
/*     Module Number: 1.2.3.5.2.1.1.1                            */
/*     Function: This module will sort the first location of     */
/*               each job in memory and place the indicies       */
/*               values in the array called order.               */
/*                                                               */
/*     Calling Modules: chk_space                                */
/*     Modules Called: None                                      */
/*                                                               */
/*     Global Variables Used: order, begin_address, number_jobs  */
/*     Global Variables Changed: order                           */
/*                                                               */
/*     Author: Ronald K. Miller                                  */
/*     System: VAX 11/780, VMS O/S and UNIX O/S: testing only    */
/*                                                               */
/*****************************************************************/

sort(order)
  int order[];
begin
  int i,j,k,l,lowest,count,temp[MAXJOBS];
  i = k =0;
  j = 1;
  count = number_jobs;
  lowest = 0;
  for(l=0;number_jobs-1;l++)
    temp[l] = begin_address[l];
                        /* Place the array of beginning address */
                        /* into a temporary array called temp.  */
  while (k < number_jobs)
  begin
    while (j < number_jobs)
    begin
      if(temp[j] < temp[lowest])
                        /* Find the lowest memory location.     */
        lowest = j;
      j++;
    end
    order[i++] = lowest;
    temp[lowest] = TOP_ADDRESS;
                        /* Make the smallest location the       */
                        /* largest value possible               */
    lowest = 0;         /* Start from the top of the array again*/
  end
  k++;
```

```
    order[k] = MAXJOBS;    /* Place the TOP_ADDRESS in the last    */
    return;                /* row and then return.                 */
  end
```

```
/*****************************************************************/
/*                                                               */
/*                          SEND_FILE                            */
/*                                                               */
/*     Date: 14 Sept 1983                                        */
/*     Version: 1.1                                              */
/*                                                               */
/*     Name: send_file                                           */
/*     Module Number: 1.2.3.5.2.2.3.1                            */
/*     Function: This module will transmit a file to the user    */
/*               or the printer depending on what was requested.*/
/*               This file has already been placed into main     */
/*               memory.                                         */
/*                                                               */
/*     Calling Modules: process_scheduler                        */
/*     Modules Called: None                                      */
/*                                                               */
/*     Global Variables Used: None                               */
/*     Global Variables Changed: None                            */
/*                                                               */
/*     Author: Ronald K. Miller                                  */
/*     System: VAX 11/780, VMS O/S and UNIX O/S: testing only    */
/*                                                               */
/*****************************************************************/

send_file(port_num)
  int port_num;
begin
  int  start;
  char value;
  start = pcb[portno].offset_address;
                                /* Inialize start to the beginning*/
                                /* address in main memory.        */
  value = inp(start);           /* Take the first character from  */
                                /* memory.                        */
  while( value != EOF)
  begin
    while((inp(ports[port_num].statport) &
                  ports[port_num].sendbit) == 0);
    outp(ports[port_num].dataport,value);
    value = inp(++start);
  end
  return;
end
```

```
/*****************************************************************/
/*                                                               */
/*                        DEALLOCATE_SPACE                       */
/*                                                               */
/*     Date: 7 Oct 1983                                          */
/*     Version: 1.0                                              */
/*                                                               */
/*     Name: deallocate_space                                    */
/*     Module Number: 1.2.3.5.2.2.3.3                            */
/*     Function: This module frees the area of main memory that  */
/*               a completed process was occupying.  This is     */
/*               done by shifting the address a row up and       */
/*               decrementing the number of jobs on the system.  */
/*                                                               */
/*     Calling Modules: clean_up, process_scheduler              */
/*     Modules Called: None                                      */
/*                                                               */
/*     Global Variables Used: portno, begin_address,             */
/*                            end_address                        */
/*     Global Variables Changed: begin_address, end_address      */
/*                                                               */
/*     Author: Ronald K. Miller                                  */
/*     System: VAX 11/780, VMS O/S and UNIX O/S: testing only    */
/*                                                               */
/*****************************************************************/

deallocate_space(row)
  int row;
begin
  int i;
  for(i=row;number_jobs-1;i++)
  begin
     begin_address[i] = begin_address[i+1];
                                  /* Shift the beginning and   */
                                  /* ending address up one row */
     end_address[i] = end_address[i+1];
  end
number_jobs--;
return;
end
```

Appendix F

AMOS Users' Guide

This is the user guide for AMOS. It covers the procedures to follow for interfacing with the operating system. The procedures covered are:

1. Log-in
2. Log-off
3. User help
4. System changes
5. User commands

Log-in

The procedure to log-in consist of sending a carriage return (<CR>). This can be done by either typing in a line of text ended by a <CR> or by entering a single <CR>. The system will prompt the user for a username. This prompt will be:

USERNAME:

The user must input their username followed with a <CR>. The system will prompt the user for a password.

This prompt will be:

PASSWORD:

The user must input their password followed with a <CR>. If both username and password are valid, the system will return the following message:

Log-in complete...

If either the username or password is invalid, the system will return the following message:

ILLEGAL USER

Log-off

For the user to terminate interaction with the operating system, the user must log-out. The following message must be typed by the user to successfully log-out:

BYE<CR>

The system will respond with the following message:

Logged out...

User Help

The user can ask for system or command information. The format for the help command is:

HELP 'subject'<CR>

The 'subject' for system information can be either USERS or DEVS. The 'subject' for command information can be one of the following:

RUN
LIST
PRINT
DEL
DIR

For the system information the operating system will return the users on-line for USERS or the devices on-line for DEVS. For the command information the operating system will return the following for the respective commands:

Format: RUN FILENAME (executable file)

Format: LIST FILENAME

Format: PRINT FILENAME (nonexecutable)

Format: DEL FILENAME

Format: DIR

If the 'subject' cannot be matched with the available information, the system will respond with the following message:

No help is available for that command

System Changes

To reconfigure the system the user must be logged-in under the 'Superuser' username. The current permissable changes are adding a username and deleting a username. The following is the format for the two system changes command:

SYS ADDUSER<CR>

SYS DELUSER<CR>

In each case the system will prompt the 'Superuser' for the desired username. This prompt will consist of:

USERNAME:

The 'Superuser' must then input the username.

User Commands

The user commands are listed in the Help User section of this appendix. The following is the required format for each command:

RUN 'filename'<CR>
LIST 'filename'<CR>
PRINT 'filename'<CR>
DEL 'filename'<CR>
DIR<CR>

The 'filename' for the RUN command must be an executable file. The 'filename' for the PRINT command must be a nonexecutable file.

If 'filename' is not located in secondary memory, the system will respond with the following message:

INVALID FILENAME

If 'filename' for the RUN command is not an executable file the system will respond with the following message:

NONEXECUTABLE FILE UNABLE TO RUN

If 'filename' for the PRINT command is an executable file the system will respond with the following message:

EXECUTABLE FILE UNABLE TO PRINT

If the username of the file being accessed isn't the same as the user's username, the system will respond with the following message:

INVALID USER FOR FILE

If all the above is valid but there is not enough space in main memory for execution of the job, the system will respond with the following message:

NOT ENOUGH MEMORY SPACE TO EXECUTE NOW

If the job is too large for all of mair memory, the system will respond with the following message:

PROGRAM TOO LARGE FOR MEMORY

If no errors occur, the command will be executed. Upon completion of the job's execution, the system will respond with the following message:

Processing of job complete...

If the job's execution is aborted at any time, the system will respond with the following message:

PROCESS ABORTED, DID NOT COMPLETE

Other Error Messages

If the user does not input one of the above command formats, the system will respond with the following message:

SYNTAX ERROR

If any new commands are added to AMOS, the formats for the new commands must be documented.

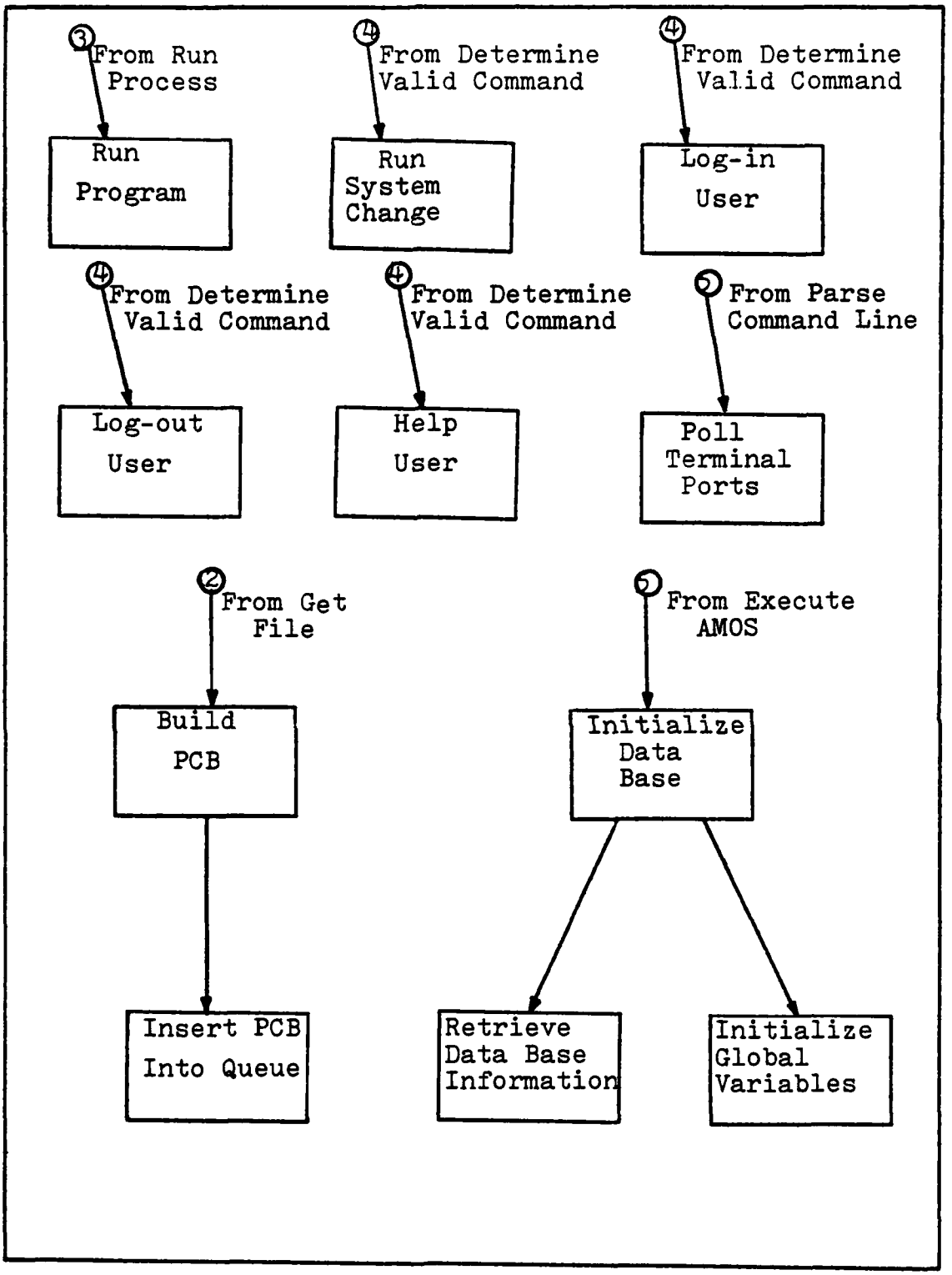
Appendix G

Hierarchical Structure of Design

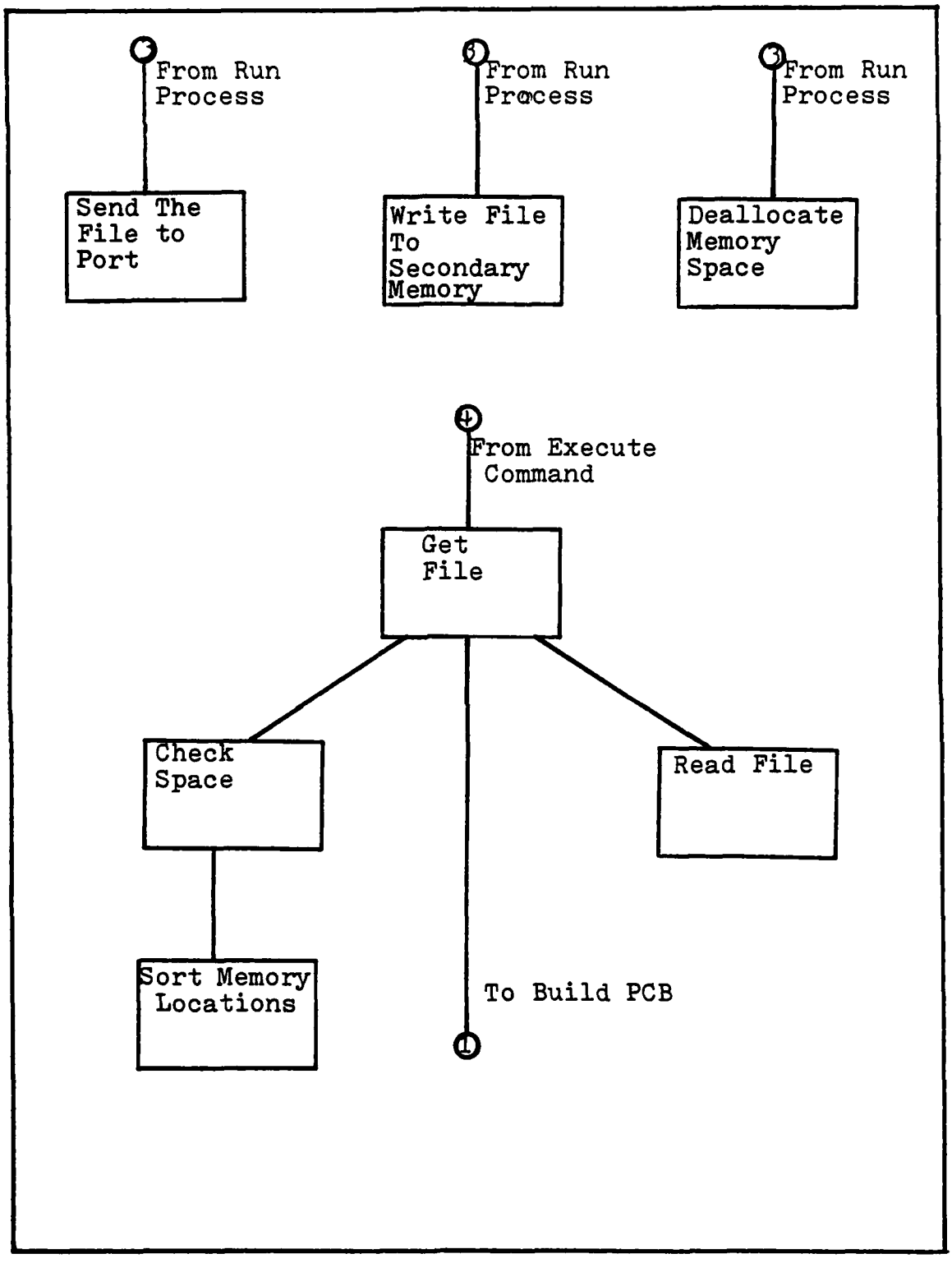
This appendix presents the overall design in hierachical form. The hierarchical concept is based on the leveling of the extended machine concept (Ref. 7: 15-20). The AMOS design can be presented in five levels that are layered around the "bare machine", or computer. The following is the five levels of the hierarchical structure presented:

1. Level 1: Lower level Process Scheduler modules and any modules directly involved in the bare machine.
2. Level 2: Memory Management Modules.
3. Level 3: Higher level Process Scheduler modules.
4. Level 4: Device Management Level.
5. Level 5: High Level Operating System Control.

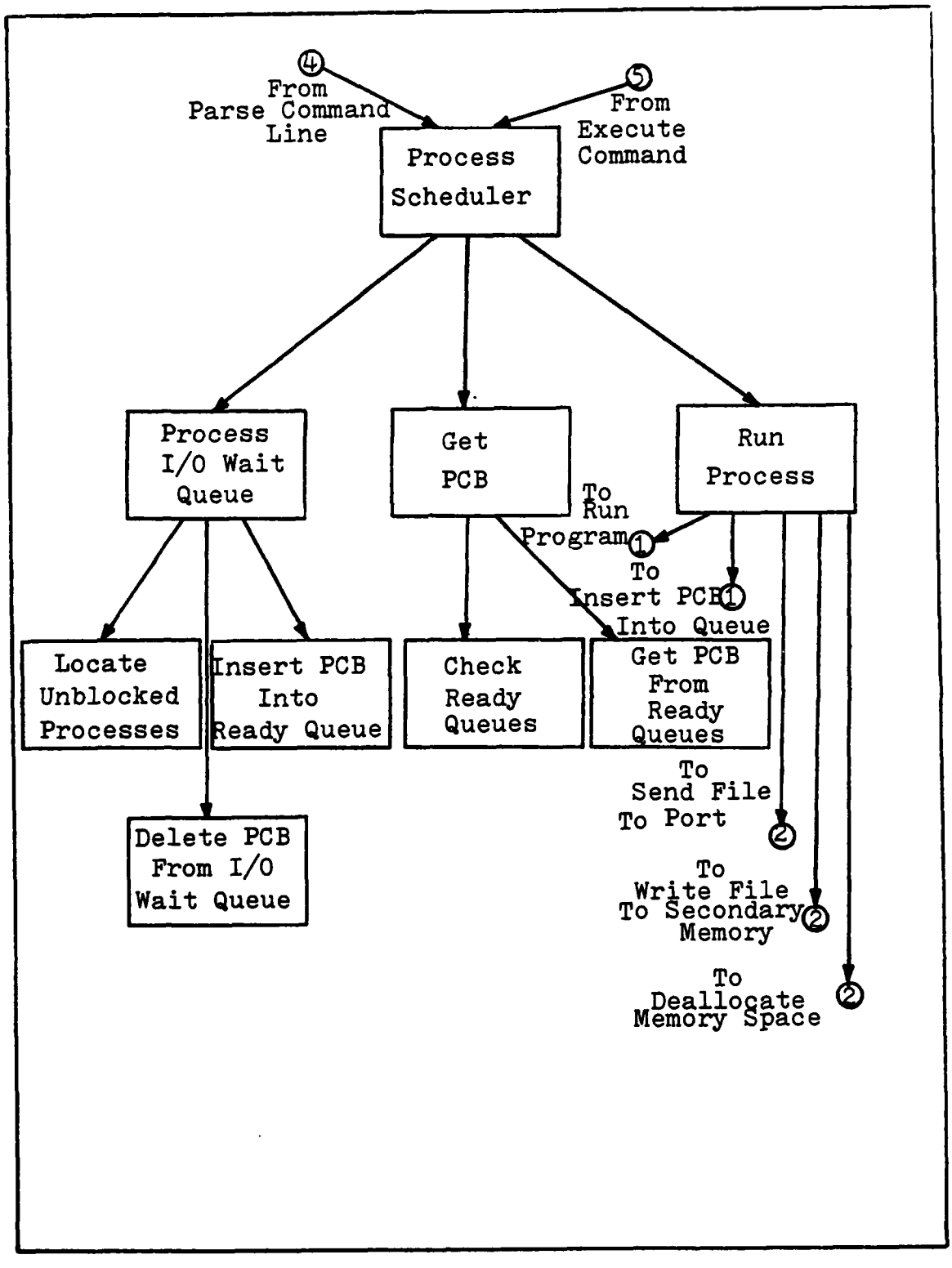
The charts present the design using the hierarchy chart. The circles with the numbers inside indicate what level the next module is located or what level the previous module is located.



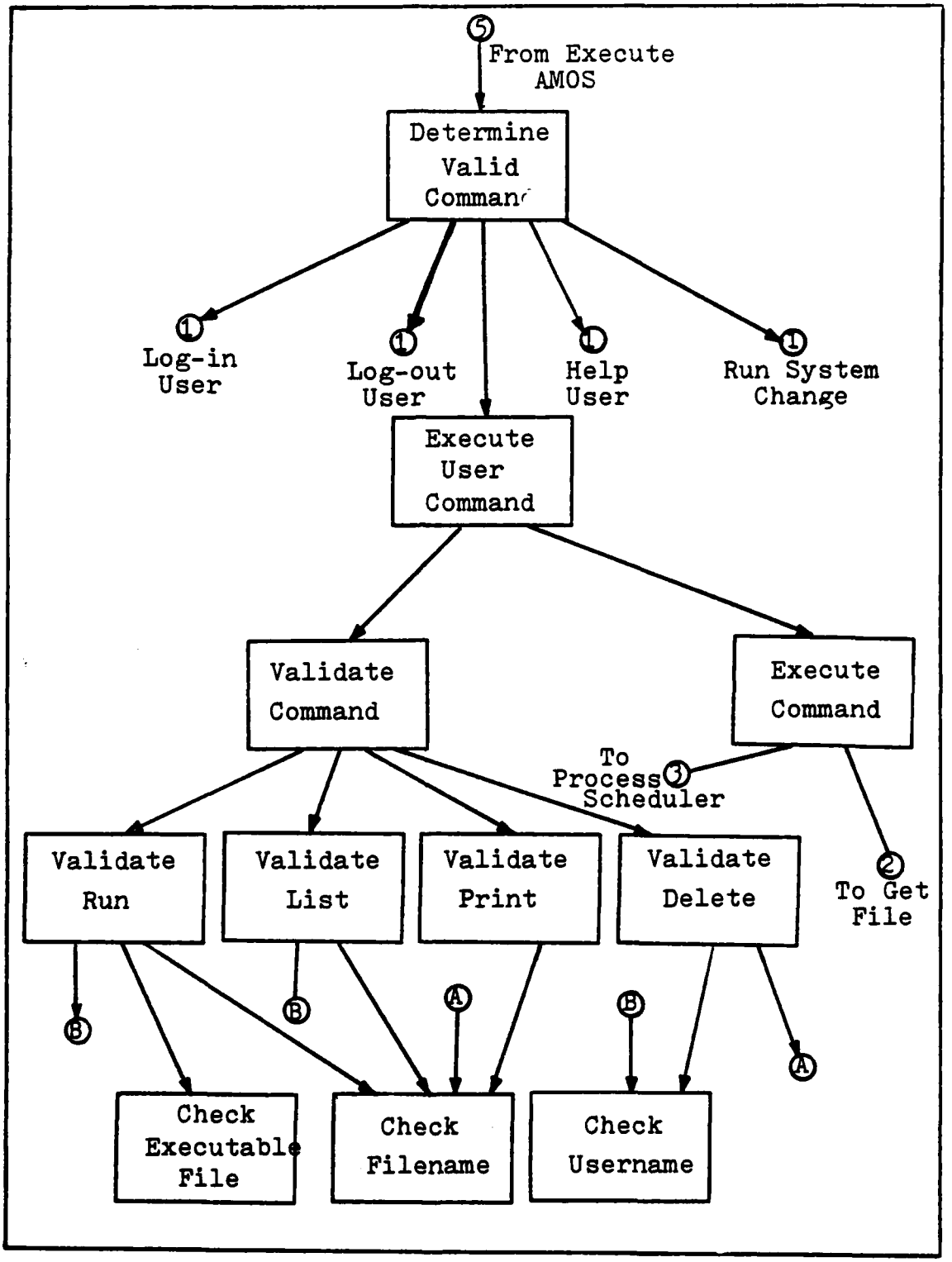
Level 1: Lower Level Process Scheduler Modules



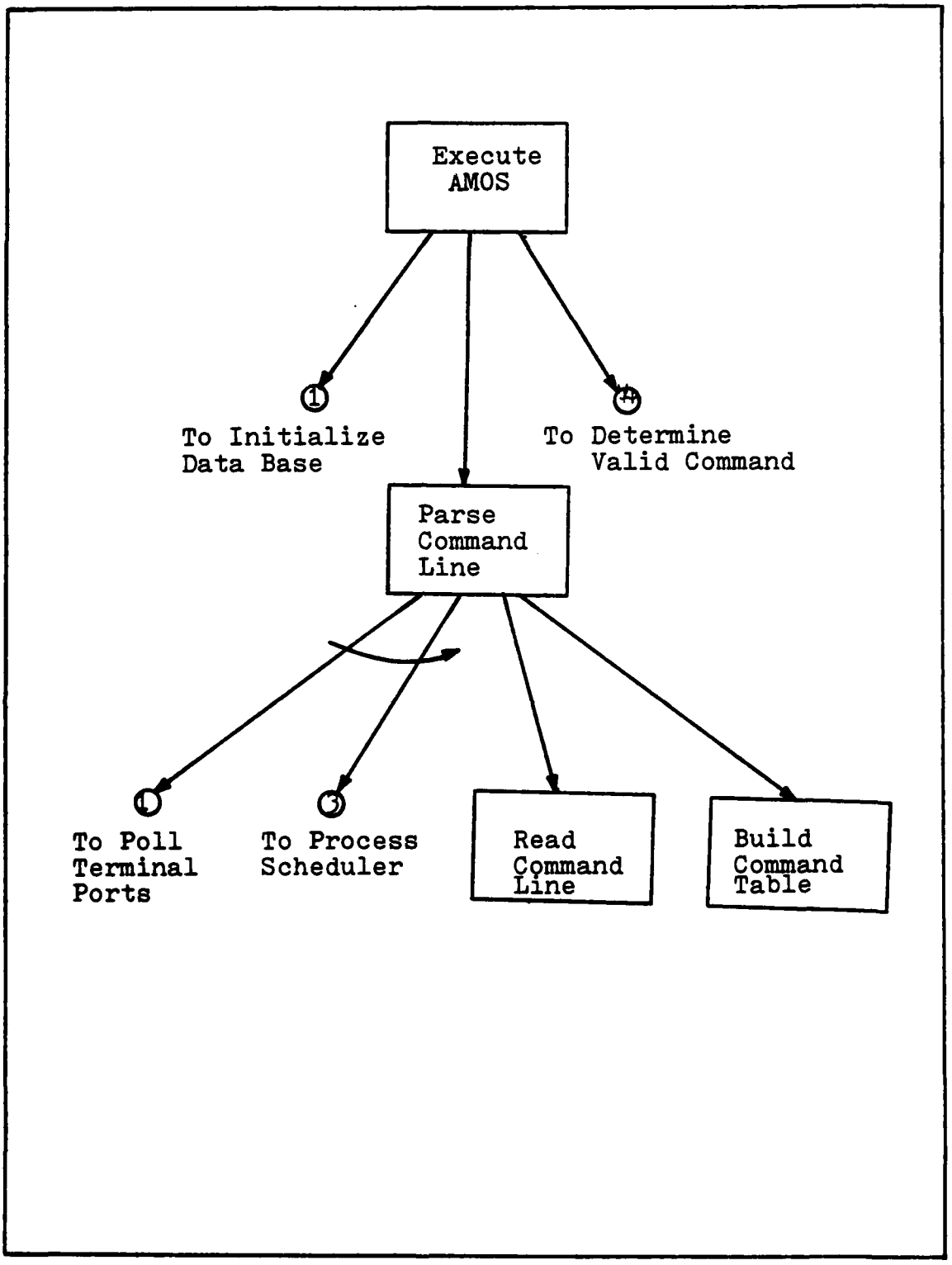
Level 2: Memory Management Modules



Level 3: Higher Level Process Scheduler Modules



Level 4: Device Management Level



Level 5: High Level Operating System Control

Vita

Lt. Paul Eugene Cruser was born on 25 December 1960 in Greensburg, Indiana. He graduated from Jennings County High School, North Vernon, Indiana, in 1978. He attended Indiana State University from which he received a Bachelor of Science degree with majors in Mathematics and Computer Science in 1982. He was commisioned upon graduation and entered the Air Force Institute of Technology in June 1982 as a first assignment.

Permanent Address:
RR# 3 Box 150
North Vernon, IN 47265

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): AFIT/GCS/EN/83D-5	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: School of Engineering; 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State and ZIP Code): Air Force Institute of Technology, Wright-Patterson AFB, Ohio 45433	7b. ADDRESS (City, State and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State and ZIP Code)	10. SOURCE OF FUNDING NOS. — PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT NO.
11. TITLE (Include Security Classification): See Box 19	

12. PERSONAL AUTHOR(S): Paul E. Cruser, 2d Lt., USAF

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
MS Thesis	FROM _____ TO _____	1983 December	252

16. SUPPLEMENTARY NOTATION

LYNN E. WOLAVER 7 Feb 84
Dean for Research and Professional Development
Air Force Institute of Technology (AIC)
Wright-Patterson AFB, OH 45433

17. COSATI CODES		
FIELD	GROUP	SUB. GR.
09	02	

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number): Operating Systems, Multiprogramming, Process Scheduling, Job Scheduling

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Title: DETAILED DESIGN AND IMPLEMENTATION OF A MULTIPROGRAMMING OPERATING SYSTEM FOR SIXTEEN-BIT MICROPROCESSORS

Thesis Chairman: Dr. Gary B. Lamont, Professor, AFIT

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ OTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL: Dr. Gary B. Lamont, Professor, AFIT	22b. TELEPHONE NUMBER (Include Area Code): 513-255-3576 — 22c. OFFICE SYMBOL: AFIT/EN

DD FORM 1473, 83 APR — EDITION OF 1 JAN 73 IS OBSOLETE.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

Abstract:

A multiprogramming operating system, designated AFIT Multiprogramming Operating System (AMOS), for the AFIT Digital Engineering Laboratory was designed at the detailed level and fully implemented, except for the assembly language routines. The requirements were developed in the works of Yusko, Ross, and Huneycutt.

This thesis effort was done in conjunction with the effort of Lt. Ronald K. Miller. This effort covers the detailed design and implementation of the overall system and, also, covers the detailed design and implementation of the operating system scheduler.

END

FILMED

3-84

DTIC